TABLES AND ARRAYS - LISTS OF SIMILAR DATA

SCHOOL OF COMPUTER TRAINING

PROGRAMMING IN BASIC
STUDY UNIT 9

TABLES AND ARRAYS — LISTS OF SIMILAR DATA

Edition 2

STUDY UNIT 9

YOUR LEARNING OBJECTIVES

WHEN YOU COMPLETE THIS UNIT, YOU WILL BE ABLE TO:

☐ Store data in the form of a table which allows master files to be created, accessed and maintained . . **Pages 1–2**

☐ Set the size and dimensions of a table. This is necessary before it can be used in a program . . . . . **Pages 3–4**

☐ Put values into a table and load them into storage . . . . **Pages 4–5**

☐ Use the NOT function to express a negative condition . . . . **Page 5**

☐ Control the repetition of program instructions with FOR...NEXT loops . . . . **Pages 5–8**

☐ Display table data on the screen via a PRINT statement . . . . **Page 8**

☐ Make the FOR...NEXT loop more versatile with the addition of the STEP clause . . . . **Page 8–9**

☐ Expand values in a table (by accessing) when calculations are involved . . . . **Pages 9–11**

[illegible] ..NEXT loops in a [illegible] The LEN (length) [illegible] ple, could fill the [illegible] no matter how [illegible] contain . **Page 12**

☐ Redefine string and numeric data through the use of the STR$ and VAL functions . . . . **Pages 13–15**

☐ Make the computer operate in either the SLOW or FAST modes . . . . **Pages 15–16**

☐ Do a lot of processing with a relatively few instructions by using "nexted" FOR...NEXT loops . . . . **Pages 16–18**

☐ Store string data as a table . . . . **Page 19**

☐ Use substringing (the slicing technique) to divide string data into pieces . . . . **Pages 22–23**

☐ Understand how string data, variables or constants can be combined through concatenations . . . . **Pages 22–23**

☐ Scan and replace strings in BASIC to do word-processing . . . . . **Pages 24–25**

☐ See how searching tabular numeric data is done by reviewing a demonstration of weekly payroll application . . . . **Pages 25–39**

☐ Store computer programs on a cassette so that they can be retrieved and used later . . . . **Pages 39–41**

[illegible] #1 . . . . **11**
[illegible] #2 . . . . **18**
[illegible] #3 . . . . **42–44**

EXAM 9 (Examination for Study Unit 9) **53–56**
ANSWER SHEET **57**

STUDY UNIT 9

TABLES AND ARRAYS — LISTS OF SIMILAR DATA

DO YOU KNOW?

- **How the use of tables can "store" data for future use?**
- **What we mean by the alternating format of a table?**
- **How a "FOR...NEXT" loop works?**

STORING "BREAD AND BUTTER" INFORMATION

Just imagine how humdrum your existence would be if you had to figure out "by hand" the payroll for 50 employees each week using tax withholding tables, social security schedule, bonus schedule, and time cards. Even using an electronic calculator with memory would not remove the dull routine of performing the same tasks over and over again.

One source of frustration is having to look up the same information time and again from various tables and schedules. A timesaver would be to transfer all the types of information onto an employee's record card so you know the tax rate, percentage of withholding for retirement, bonus credits, etc.

With employee turnover and the various other changes occurring within a company, however, it is difficult to maintain accurate, up-to-date records. Each time there is a change of employee or employee status, the change has to be made on several records. Small wonder, then, that computer programmers have devised

FIGURE 1—Engineers today rely upon a computer's "instant answers" to construction questions. Whether it is having to do with volume, weight, tensile strength, area or mass, there are programs, tables and arrays readily available—thanks to programmers.

ways to remove the tedium and hand labor from such tasks.

Today, many companies store payroll records on computer. There are programs which automatically change the payroll records with the touch of a few keys on the computer console.

Perhaps the greatest timesaver of all, however, is the computer's ability to store vast quantities of data such as income tax tables, bonus schedules, etc. This variable data is just waiting to be used in figuring the payroll. Want to know the withholding amount for an employee earning $500 with two dependents? The answer is instantly available because the tax table is stored in computer memory.

Can you believe that you can design programs which are capable of storing more data than your computer can hold at any one time? Thanks to auxiliary storage in the forms of magnetic tape and disks, it is possible. Programs and data can be stored on shelves until you and your clients are ready to use them.

The ability of programmers to devise ways of storing vast quantities of data, such as federal income tax withholding tables, has made a tremendous impact upon record keeping. Not only that, such storehouses of data make it possible for the programmer and user to play "What if?" type games and make projections.

"What if, for example, we increased the bonus structure by 20%? How much would it cost the company?" Answers to such questions are waiting—provided the right program and data are in storage.

You will learn how to set up tabular data files in this lesson. The applications of such programs are almost limitless. Let's find out more about how to make humdrum jobs a breeze—using your know-how.

TABLES

Simple variables are capable of containing one value at a time. Values can be given to variables either by the INPUT statement or by a LET statement.

It is often desirable, in data processing, to store different values— simultaneously—under the same variable name. When this is done, a "table" has been created.

Suppose, for example, that we wished to store a series of test grades for a student so that we can, any time the program is executed, determine the student's average grade. We also might desire to find out what the student's highest score is at any time. Our grade score table would look like this in main storage:

TEST SCORES

9Ø	85	78	95	84	88	92	1ØØ	96	82

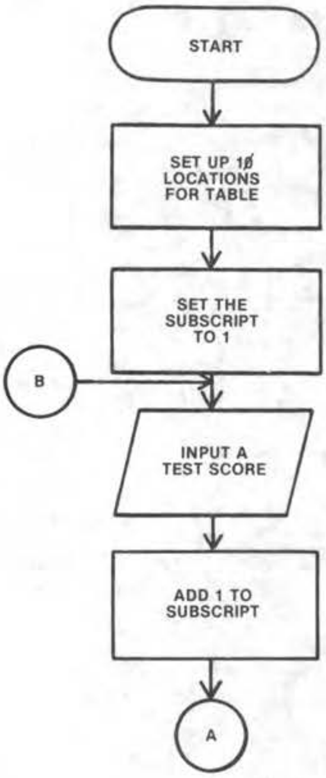
The definition of a table is: a series of bytes containing values located in consecutive positions of main storage. The table itself is defined by assigning a variable name. Each of the various values within the table—the elements, if you like—can be addressed through the use of the variable name and a subscript.

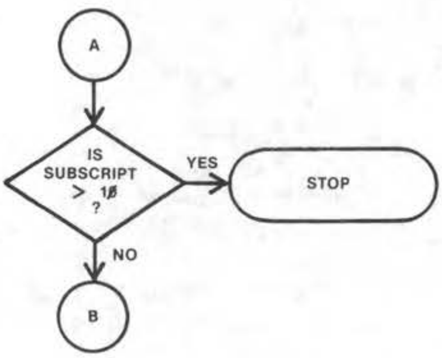
Now, you ask, what's a subscript? A subscript is a literal (an actual number) or a numeric variable which points to a particular element within the table. For example, if we called our test score table "T," then each of the test scores in the table could be referenced as follows:

T (1) = 9Ø
T (2) = 85
T (3) = 78
T (4) = 95
T (5) = 84
T (6) = 88
T (7) = 92
T (8) = 1ØØ
T (9) = 96
T(1Ø) = 82

Each element of the table must be described by showing the table's name (T), and, enclosed in parentheses, the subscript which points to a particular element in the table. Each element can also be referenced by a variable subscript.

For example, if "A = 3," then the "T(A)" is the third element of our test score table (78). In order to enter each test grade into the computer, we would initially employ the following design logic:





PROGRAMMER'S REMINDER
A WORD ABOUT FLOWCHARTS...

Flowcharts are to programmers what sketches are to artists. Flowcharts are merely design tools. They are used to sketch the major logic decisions and paths the program will take.

Experienced programmers do not include every detailed instruction in their flowcharts. Such things as printing prompts are sometimes left out, for example, because such prompts are understood to exist whether sketched or not.

You may notice, in this lesson and in other references, flowcharts which appear to be incomplete. Some minor details will be missing. However, the major logic must *always* be included or else program documentation is faulty and incomplete.

Once you have had sufficient practice in designing programs, you will recognize which elements can be eliminated from your flowcharts. Meanwhile, it is wise to include *all* details so they become thoroughly understood.

DIMENSIONING THE TABLE

When it becomes necessary to store more than one value under the same variable name, the variable must first be dimensioned as a table. The DIM statement is used to do this. To save our 1Ø test grades, our first BASIC statement might read:

```
1Ø DIM T(1Ø)
```

This will reserve 1Ø consecutive locations in RAM, each one of which can hold one value. All 1Ø values will also be initialized to zero.

To demonstrate this, do the following:

ENTER the statement:
1Ø DIM T(1Ø)

RUN the program. . .then,

ENTER the command:

PRINT T(1Ø)

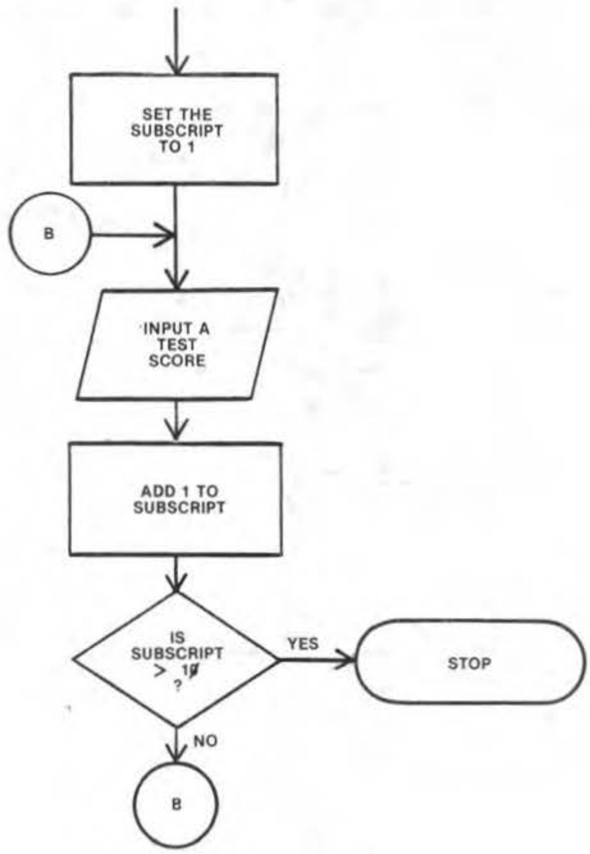
> **PROGRAMMER'S REMINDER**
>
> **The value Ø appears at the top of our screen; in other words, the tenth element was set to Ø by the DIM statement. Try printing the rest of the elements (i.e. PRINT T(1), T(2), etc.); they will all equal Ø. It is important that we DIM our tables *before* we put any data into them. If we were to DIM it later, any previously stored values would be "clobbered" (that is, set back to Ø).**

LOADING THE TABLE

Once space in main storage has been set aside for our table, we next need to enter our test scores. We could use a series of INPUT statements such as:

```
2Ø INPUT T(1)
3Ø INPUT T(2)
4Ø INPUT T(3)
 }     }     }
11Ø INPUT T(1Ø)
```

But, rather than this tedious approach, we can set up a variable subscript, which we can initialize to one and increment by one until we have entered all 1Ø test scores. Following our flowchart design, we would have:




```
1Ø DIM T(1Ø)

2Ø LET A = 1

3Ø PRINT AT 21,Ø; "ENTER SCORE:"

4Ø INPUT T(A)

5Ø LET A = A + 1

6Ø IF NOT A > 1Ø THEN GOTO 4Ø

7Ø STOP
```

Line 2Ø sets the subscript to one so that the first value entered will be stored in the first position of the table. After issuing a PRINT at the bottom of our screen (statement 3Ø), we encounter our by now familiar INPUT statement (line 4Ø).

Notice, however, that rather than inputting a simple variable (T), we must enclose the subscript in parentheses so that the computer will know where to store the grade entered.

The subscript is then incremented by one (line 5Ø). The IF statement on line 6Ø tests to see if we have already entered our 1Ø values. If so, the loop terminates; otherwise, a branch occurs back to line 4Ø where the next value will be inserted into the next location of the table.

Enter this program and RUN it. Input the 1Ø test scores. When the program terminates its loop, use the PRINT command to display the third test score entered ((i.e. PRINT T(3)). The value "78" should appear on your display as this was the third element of the table.

Now, try to display the eleventh element (as in PRINT T(11)). A report code will be returned, indicating that the subscript is out of the range of the table. When using a table, it is of utmost importance that the programmer ensure that subscripts be greater than zero and not greater than the dimension of the table.

THE *NOT* FUNCTION

You may have noticed the use of the NOT function on line 6Ø. Here it has been used as a substitute for coding:

```
6Ø IF A< = 1Ø THEN GOTO 4Ø
```

The relational operator, "NOT", provides a convenient way of testing for one condition instead of two. This chart shows how NOT can be used and the expressions it replaces:

USING NOT......	REPLACES...
NOT A < B	A > = B
NOT A > B	A < = B
NOT A = B	A < > B

FIGURE 2—Every aspect of the transportation industry is computerized—from ticketing air passengers to keeping track of freight cars and shipments. Not only that, tables and arrays can be constructed to report comparative data in terms of volume, profits, costs, etc. A clever programmer is able to adapt larger programs used by industry to fit the needs of smaller companies.

"FOR ... NEXT" LOOPS

The variable "A" as used in our program is critical to the execution of our program. Not only does it allow our 1Ø values to be loaded into the table (by adding one to it), but it also controls the number of loops until the program is ended (1Ø).

There is another method used in BASIC programming to cause a fixed number of loops (or iterations). This is the "FOR...NEXT" loop.

Modify the first program to look like this:

```
1Ø DIM T(1Ø)
2Ø PRINT AT 21,Ø; "ENTER SCORE:"
3Ø FOR A = 1 TO 1Ø
4Ø INPUT T(A)
5Ø NEXT A
7Ø STOP
```

Try running this program with the data we used before. Use the PRINT command to display some values from the table. It should produce the exact same results. How does it do this?

The FOR statement on line 3Ø established "A" as a control variable. "A" is initially assigned a value that is on the right side of the equals sign (=).

The number that appears to the right of the keyword "TO" gives the last value "A" will contain before the loop is ended.

A "FOR" statement must be used in conjunction with a NEXT statement, which follows it in the program.

The "NEXT" statement has only one operand beside it: the control variable "A." However, it causes some powerful work to be done.

The "NEXT" statement increments the control variable by a fixed amount (in this case, one) and then tests to see if the upper limit given in the "FOR" statement has been reached. If it hasn't, a branch occurs back to the statements within the "FOR...NEXT" loop. If the maximum has been exceeded, then the control loop ends with the execution continuing with the instruction after the "NEXT" statement.

FIGURE 3—A drill press is one of hundreds of tools which can now be driven and controlled by computer programs. Among the considerations for such programs are types and densities of materials to be drilled, types and sizes of bits, etc. Programming may depend upon the use of tables in order to select the proper bit, drilling speed, and amount of pressure to be applied.

FORMAT

nn FOR v = x TO y

nn NEXT v

where: nn are line numbers,
v is the control variable,
x is the initial value assigned to v
and y is the upper limit of v.

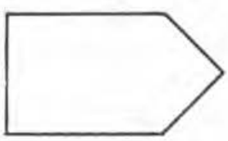
The instructions coded within the "FOR...NEXT" loop can be extensive and varied, but here are two things you should avoid doing:

1. It is dangerous to branch *outside* the "FOR. . . NEXT" loop via a "GOTO" statement unless you are careful to branch *back* so that the "NEXT V" line is encountered. Otherwise, the computer may lose control of your fixed loop. For this reason, "GOSUBs" are much safer to use since they always RETURN.

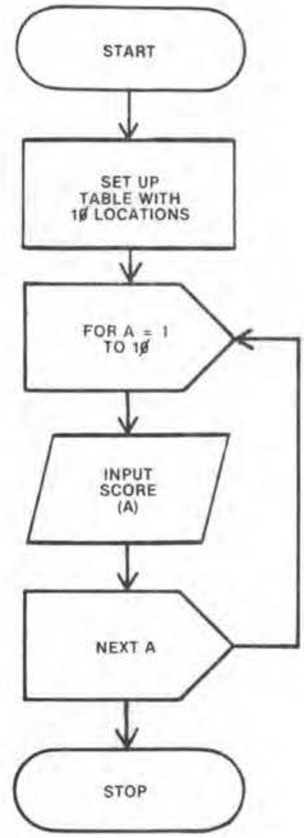
   Branching *within* the loop is, of course, a useful and sometimes necessary tool when you wish to bypass or repeat some steps within the loop.

2. Be especially careful in how you use the control variable. In our example, the control variable was safe to use as the subscript of the table as well. But assigning other values to the control variable (as in a "LET" or "INPUT" statement) can create havoc with a "FOR. . . NEXT" loop.

   The "FOR. . . NEXT" loop is not an easy sequence to design using traditional flowcharting symbols. The flowchart we used to load the table is equally valid for both methods (using the "IF. . . THEN" and "LET" statements or the "FOR. . . NEXT" instructions). But so much happens in our second method that some designers choose to employ a new symbol to demonstrate fixed loops. This symbol is a combination of the processing symbol (▭) and the decision box (◇).



This control box is used at the top and bottom of the "FOR. . . NEXT" loop. Our test score loading program could be designed as follows:



Whichever method you use, the same steps occur. That is,

1. . . .the control variable will be initialized to the starting value;

2. . . .the statements following the "FOR" statement will be done;

3. . . .the "NEXT" statement will increment the control variable;

4. . . .a test will be made to determine if this value of the control variable is greater than the maximum;

5. . . .if the value is less than or equal to the maximum, the steps inside the loop are repeated;

6. . . .if the value is greater, the control loop has ended and the instructions following the "NEXT" statement will be executed.

PRINTING THE CONTENTS OF A TABLE

Once we have loaded values into a table, we can find many ways to manipulate that data. First, we can display the contents of the data by establishing another "FOR...NEXT" loop — this time with a "PRINT" statement. For example, add these statements to your test score program:

```
 6Ø CLS
 7Ø PRINT "TEST SCORES"
 8Ø FOR V = 1 TO 1Ø
 9Ø PRINT T(V)
1ØØ NEXT V
11Ø STOP
```

Your program will now display the test grades you entered. Notice that the subscript chosen this time (and the control variable) was "V." We could have used "A" again, but we will be much safer in a longer program if we choose unique variables as the controllers of "FOR...NEXT" loops.

Before you enter the "PRINT" command that follows, think about what the results should be. What is the value of "V" when the program was terminated? Enter the following statement:

```
PRINT T(V)
```

What happened? Your computer returns a report code indicating that the subscript is out of range. Now, enter this command:

```
PRINT V
```

Notice that the number "11" is displayed. Since our table was dimensioned at "1Ø", there is no eleventh value. But the control loop was terminated when the control variable *exceeded* the upper limit. Notice how we have used the PRINT command as a debugging tool.

Whenever your program terminates abnormally (unless you have lost it altogether), commands such as PRINT can help to show what went wrong and usually can aid the programmer in eliminating the "bug."

"GET MORE DATA!"

FIGURE 4—Slumping sales can cause absolute mayhem inside the company. The call will go out for "more data!," and you may find yourself coming to the rescue. Perhaps the program simply did not ask all the right questions about sales, production, and profits.

For example, when a program produces surprising (and incorrect) results, PRINTing the contents of the variables used to generate these results could prove to be an invaluable debugging tool.

FOR SAFETY!

In long programs, it is best to use unique variables as the controllers of "FOR...NEXT" loops.

THE STEP CLAUSE

The "For...NEXT" loop can also increment the control variable by values other than one by appending the STEP clause. If, for example, we wanted to see every other test score in our pro-

gram (the 1st, 3rd, 5th, 7th and 9th scores, only), we can change line 8Ø to read:

8Ø FOR V = 1 TO 1Ø STEP 2

The "STEP 2" now causes two to be added to the control variable "V" every time the "NEXT V" statement is encountered.

Insert this change into your program. But this time, *don't* "RUN" it. The RUN command causes all of your data to be wiped out so that you will have to INPUT your test scores again. Instead, enter this command:

GOTO 6Ø

You should see every other score displayed from the table on the top of the screen. The GOTO command is another useful tool in executing and debugging a program—especially when a lot of data has already been entered into a table. Keep this in mind: *the GOTO command does not initialize any values.* They will be left at the values they contained previously.

When a negative number appears to the right of the STEP clause, the control variable is subtracted from (decremented) or reduced whenever the NEXT statement is executed.

We could have our test scores printed in the reverse order from the way they were entered by again modifying line 8Ø. Change the line to read as follows:

8Ø FOR V = 1Ø TO 1 STEP — 1

This time, the control variable is initially set to 1Ø. The tenth score will be displayed. Then, one will be subtracted from "V" repeatedly so that the ninth through the first elements are displayed from our table. Execute this modification via another "GOTO 6Ø" command. You should see the test scores listed from last to first.

If you now use the command "PRINT V," you will see the value "Ø" displayed. When using a negative increment STEP, the loop terminates when the control variable is *less than* the value to the right of the "TO."

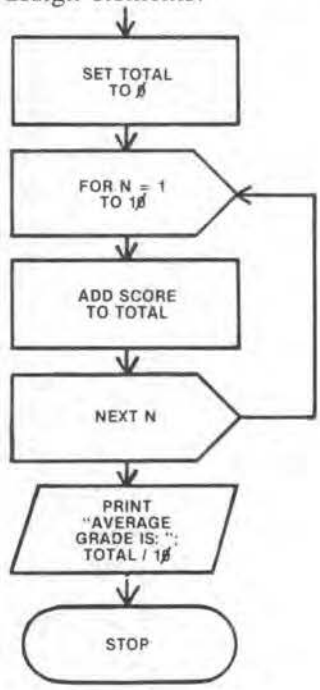
PROGRAMMER'S REMINDER

The STEP clause is not needed whenever we wish to increment by one. In other words, in this circumstance, STEP 1 is assumed.

CALCULATING VALUES IN A TABLE

Let's now get back to our original program. Change Line 8Ø back to read 8Ø FOR V = 1 TO 1Ø. Suppose we wanted to calculate the average of the grades entered. It is easy to add the logic necessary for such calculations.

After loading and printing the test scores, we can use a "FOR. . .NEXT" loop to add them together. When we have added up all 1Ø scores, dividing the total by 1Ø will give us the average. Here are the design elements:



11Ø LET T1 = Ø

12Ø FOR N = 1 TO 1Ø

13Ø LET T1 = T1 + T(N)

14Ø NEXT N

15Ø PRINT "AVERAGE GRADE IS: "; T1/1Ø

16Ø STOP

First, an accumulator (T1) is set to "Ø." Then, a "FOR...NEXT" loop is encountered which uses the control variable "N." Statement 13Ø adds that test score from the table (T), which is in the position pointed to by the subscript ("N" again). When all 1Ø scores have been totaled, the PRINT instruction will display the average. The new program will now look like this:

```
1Ø DIM T(1Ø)
2Ø PRINT AT 21,Ø; "ENTER SCORE:"
3Ø FOR A = 1 TO 1Ø
4Ø INPUT T(A)
5Ø NEXT A
6Ø CLS
7Ø PRINT "TEST SCORES"
8Ø FOR V = 1 TO 1Ø
9Ø PRINT T(V)
1ØØ NEXT V
11Ø LET T1 = Ø
12Ø FOR N = 1 TO 1Ø
13Ø LET T1 = T1 + T(N)
14Ø NEXT N
15Ø PRINT "AVERAGE GRADE IS: ";
    T1/1Ø
16Ø STOP
```

Reload the scores and RUN the program. You will see that the average of the grades entered is 89.

Until now, we have assumed that each time we run the program, we will have 1Ø grades to enter. What if we don't know how many scores we will be entering for each run?

All that is required is to establish the size of the table at a maximum value *before loading the scores!* Then, each time grades are entered, we can insert logic to count the number of scores and use this value for displaying and averaging purposes.

The only problem is that the "FOR... NEXT" statements surrounding the INPUT instructions seem to be set for a fixed number of loops (1Ø). We will need a way to *force* the loop to end before the maximum number of scores is entered. At the same time, we will wish to maintain control by *not* using a GOTO instruction that leaves the loop.

We need to establish a maximum number. Let's say that 5Ø test scores are the maximum that we wish our program to handle. Lines 1Ø and 3Ø would need to be changed as follows:

```
1Ø DIM T(5Ø)
3Ø FOR A = 1 TO 5Ø
```

By making this change, we will be able to load 5Ø scores into the table. However, we must tell the table to accept *fewer* than 5Ø scores and still keep track of how many scores were entered.

Since we can assume that no test can have a grade of more than 1ØØ, we can have the user enter a grade greater than 1ØØ when all grades have been given. Insert the following logic:

```
32 INPUT G
34 IF NOT G > 1ØØ THEN GOTO 4Ø
36 LET A = 5Ø
38 GOTO 5Ø
4Ø LET T(A) = G
42 LET T2 = T2 + 1
```

This time, when we are prompted for a grade, the value is not automatically placed in the table. Instead, it is first compared to 1ØØ.

As long as the score is not greater than 1ØØ, the LET statement on line 4Ø puts the score into the table in the position specified by "A" and one will be added to the counter, "T2."

What happens when a score is greater than 1ØØ? The control variable will be forced to equal its maximum and a branch will occur to line 5Ø, "NEXT A." This will result in "A" being incremented by one, thus terminating the loop. Notice that our GOTO statements on lines 34 and 38 do not exit from within the "FOR...NEXT" loop.

Before we run the program, we have to make a few more changes. The accumulator which keeps track of the number of grades entered (T2) has to be set to "Ø." Add the following line:

```
25 LET T2 = Ø
```

We must also account for the 5Ø grade values. Even though we may enter fewer than 5Ø grades, there will still be 5Ø values in the table—the ones not entered will still have a value of zero.

If we changed our PRINT and accumulating "FOR...NEXT" loops to execute 5Ø times, we would be displaying and calculating more times than is necessary.

For this reason, control these loops from one to "T2" times, so only those values entered will be used. Change lines 8Ø and 12Ø to read:

```
8Ø FOR V = 1 TO T2

12Ø FOR N = 1 TO T2
```

Then, modify the average calculation instruction to read as follows:

```
15Ø PRINT "AVERAGE GRADE IS: ";
    T1/T2
```

RUN the program as before. ENTER a grade more than 1ØØ when prompted for the eleventh time. You should have the same average as before, 89.

The difference, however, is that we can now enter as many test scores as we desire (up to a maximum of 5Ø). Try it with only two grades, 1ØØ and 9Ø. The average, of course, will be 95.

See how easy it is to control the size of the table by merely modifying the DIM statement (line 1Ø) and the "FOR...NEXT" loop on line 3Ø? You could dimension the table for 1ØØ grades, for example, just by changing these two lines.

PROGRAMMER'S CHECK

1

Dimensioning Tables

Is there an ultimate maximum to the dimensions of a table? Try to find out how large you can make the table already presented in your lesson by increasing the number in the DIM statement. Once you have discovered the "ultimate" dimensions of the table, see Page 12 for more discussion about reaching the maximum.

FIGURE 5—Computer literacy is a necessity. Students are required to take computer courses in many localities. People of all ages must learn how to apply computer technology to everyday life.

PROGRAMMER'S CHECK ANSWERS

1

Did you increase the DIM statement number until you reached the limits? The signal indicating that you've reached the maximum dimension occurs when the program is run. An ERROR message will appear when there is not enough room in RAM for both the table and the program. However, the number should be in the thousands before the ERROR signal is reached.

It is important to know what happens when you reach the limits of RAM. The smart programmer always plans ahead so there is plenty of extra space in RAM to accommodate the program and data. Filling RAM to its limits can result in not being able to "save" the input because you won't have room for additional instructions. More will be said about "running out of RAM" in the future, but keep RAM limits in mind when programming.

OTHER USES OF THE "FOR ... NEXT" STATEMENT

While "FOR...NEXT" loops provide a convenient way for manipulating tables and their subscripts, we are by no means limited in our use of them.

Fixed Loops

Whenever we wish to establish a fixed number of iterations within our program, we can use the controlled loop technique. For example, you could print your name all over the screen by running the following program:

```
1Ø CLS

2Ø FOR N = 1 TO X

3Ø PRINT "YOUR NAME";

4Ø NEXT N

5Ø STOP
```

Before you enter and run this program, you will need to establish and substitute a value for "X" (line 2Ø). To do this, determine how many times your name can appear on the screen. Exceeding this number would be no problem other than the fact that the program would terminate whenever the screen becomes full.

Consider this. You can print 32 characters on 22 different lines (32 x 22 = 7Ø4) or print positions. Divide your name into 7Ø4 and substitute this result for "X."

For example, suppose your name were JOHN. You would substitute 176 for "X" because 7Ø4 divided by 4 equals 176. When you run this program, watch how fast it works.

Now suppose we wished to create a program which would allow us to input *any* name and fill the screen with it. This would necessitate an INPUT statement, of course. But we will then have the problem of determining the number of loops, since the *length* of the name fixes the control loop. Fortunately, we have a BASIC instruction to do this.

The LEN (length) function examines a string (or a string variable) and returns the number of bytes that it contains.

FORMAT

LEN {string variable / string}

For instance, the statement

LET A = LEN "WORD"

would assign the value four to "A" since the

string "WORD" occupies four bytes of storage. Let's see how we can use this function to control our "FOR. . .NEXT" loop. RUN the following program and see how it works.

```
10 INPUT N$

20 CLS

30 LET P = LEN N$

40 LET X = 704/P

50 FOR C = 1 TO X

60 PRINT N$;

70 NEXT C

80 STOP
```

Enter RUN, then enter YOUR NAME.

Now here's another way to do the same thing.

We would merge the calculations from lines 30 and 40 into our "FOR. . .NEXT" statement by deleting lines 30 and 40 and modifying line 50 to read:

```
50 FOR C = 1 TO 704/LEN N$
```

It should work just as well. Try it and see.

PROGRAMMER'S REMINDER

The modification just performed raises an interesting programming issue. While the second method occupies less storage (one line instead of three), the first method is certainly easier to follow. When storage space becomes limited, figure out how to combine instructions. But in the initial design, coding, and testing of a program, you will probably be better off if you develop your program step by step. Debugging is much easier when each instruction does only one thing.

STR$ AND VAL

In our "screen-filling" program, we saw how the LEN function returned a value indicating the length of the string or string variable. What if we wished to know the length of a number or numeric variable? Consider these statements.

```
10 INPUT N

30 LET P = LEN N
```

These statements would not be valid, since "N" is a numeric variable. Several BASIC functions work only with strings; others work only with numbers. The STR$ and VAL functions serve to convert the definition of numerics to strings and vice versa.

STR$ is a function whose "argument" is a number or numeric variable. Examine this statement.

```
15 LET N$ = STR$ N
```

Notice how the above statement redefines the *characteristics* of the numeric variable "N" without changing its *value*. Insert this statement into your program and modify line 10 to produce the following program:

```
10 INPUT N

15 LET N$ = STR$ N

20 CLS

50 FOR C = 1 TO 704/LEN N$

60 PRINT N$;

70 NEXT C

80 STOP
```

RUN this program. You will see that it works just like the first one, except that you must enter numeric values when the prompt appears at the bottom of the screen. If you input

FIGURE 6—Whether the task is storing stock market results or projecting labor trends, the computer and custom software have become vital tools used by government, business, industry and individual citizens.

alphabetic characters, the program will not run. ENTER 2Ø2 or a longer number and see what happens. Try more number combinations. Can you play a trick on the computer by inserting a letter character into the middle of a long number? Try it for yourself.

Actually, since they contain the same values, you could print "N" instead of "N$" on line 6Ø. Try running the program by substituting:

```
6Ø PRINT N;
```

PROGRAMMER'S REMINDER

Although the program still "works," it is considerably slower. This is another example of how much more difficult it is for the computer to handle numerics than strings. Whenever possible, therefore, define your fields as character variables; they are manipulated byte by byte without having to determine the sign and size of the value. (Sign and size of values must be done for numerics.)

As we already know, many functions (including all simple arithmetic statements) can only be done for numbers. The VAL (value) function does exactly the opposite of the STR$ function. The VAL converts a string to a numeric without changing the values.

While there are no restrictions on the values that STR$ can convert, the VAL function can only handle characters that are numbers anyway. For example, try running the following short program.

```
1Ø INPUT A$
2Ø LET A = VAL A$
3Ø LET A = A + 1
4Ø PRINT A
5Ø STOP
```

RUN this program, entering any "number" when prompted for a value for "A$." Your number, plus one, will appear on the screen. The VAL function converted the string variable, "A$," to a numeric variable, "A," so that arithmetic could be performed on it.

(If we then wished this value to be considered as a string, we could insert line 35 to read:

```
35 LET A$ = STR$ A
```

and we would be back to "A$" just as if we had added one to "A$," which we would normally say was impossible!)

RUN this program again. This time, enter some letters to be used as the value of "A$." On this run, a report code will be returned. Remember, the VAL function will only work on string variables which contain numbers only!

FAST VERSUS SLOW MODE

There are two BASIC keywords which are used to control the execution of commands within your computer. Let's find out more about FAST and SLOW and how these can be used for greater operating efficiency.

Your computer is normally in the SLOW mode. In this mode, your screen is constantly displayed—even while loops and calculations are being done.

You can alter this normal, SLOW mode to the FAST mode as either a *COMMAND* or as a BASIC statement. In the FAST mode, your screen will go blank until all calculations and loops are done. Then, the screen will be displayed.

Using the FAST mode will increase the computer's execution speed up to 500%! Let's

compare the speed of these two modes while using the same program.

```
 1Ø INPUT N$

 2Ø SLOW

 3Ø FOR T = 1 TO 2

 4Ø CLS

 5Ø FOR C = 1 TO 7Ø4/LEN N$

 6Ø PRINT N$;

 7Ø NEXT C

 8Ø FAST

 9Ø NEXT T

1ØØ STOP
```

ENTER and RUN this program as before. This time, your name will fill the screen twice. The first time will be in the SLOW mode; the second time, in the FAST mode. Note the difference in the way your print display appears and how much faster it comes up the second time.

Your computer will now be in the FAST mode, which will remain in effect until you change it or shut the computer off. This may result in a jerky, unpleasant mode for the entering and editing of a program. Simply enter the SLOW command and all will be back to normal. Alternatively, insert line 95 as follows:

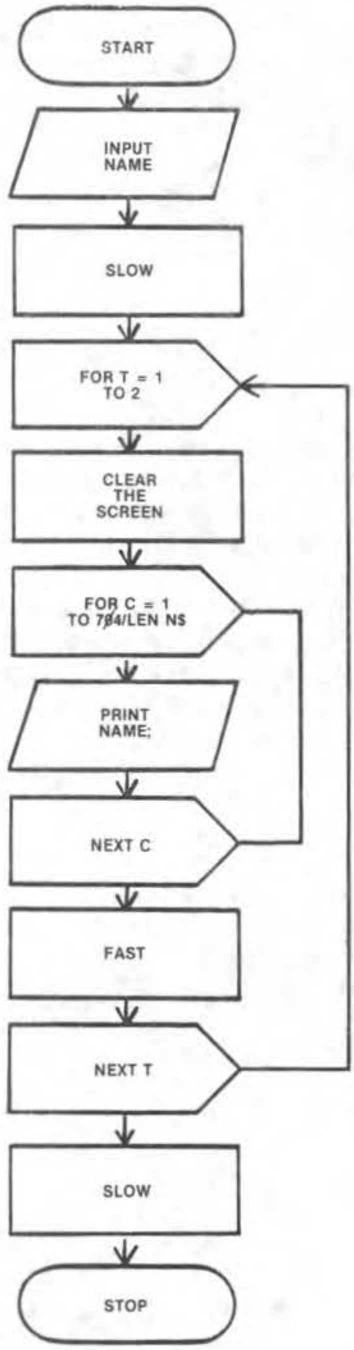
```
95 SLOW
```

Then, when your program stops executing, you will be back in the SLOW or normal mode.

NEXTED "FOR ... NEXT" LOOPS

We have snuck in an interesting addition to our name displaying program. Within a control loop, we have another control loop. This is what is meant by a *nested* "FOR. . . NEXT" loop.

A design of this program would look like this one.



The loop controlled by the variable "T" causes the output screen to be displayed twice: first in the SLOW mode and then in the FAST mode. The loop controlled by our control variable "C" displays the name on the screen.

Many levels of *nested* "FOR...NEXT" loops can be within one program. Each, however, must be assigned a unique control variable.

The next sample program demonstrates the use of *nested* "FOR...NEXT" loops in determining the square footage of a room where lengths and widths will vary. In this program, the area is equal to the length times the width (A = L*W) and the "FOR...NEXT" loops will be used to vary the room size from 1Ø to 15 feet on each side. Here is the program.

```
1Ø CLS

2Ø FOR L = 1Ø TO 15

3Ø FOR W = 1Ø TO 15

4Ø PRINT L*W,

5Ø NEXT W

6Ø NEXT L

7Ø STOP
```

ENTER and RUN this program. On your screen, you will see 36 different areas of our hypothetical room in two columns. Let's see how it works.

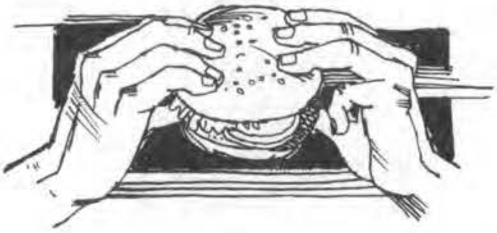


FIGURE 7—Are you or a member of your family counting calories? You can create a calorie table, measure your daily input, and develop an accurate picture of weight gains, losses, and food consumption by day, week, month and year by using your computer.

1. After the screen has been cleared, the variable "L" (length) is initialized to "1Ø."
2. The variable "W" (width) is also set to "1Ø."
3. Our first area, therefore, is displayed as "1Ø*1Ø" or "1ØØ."
4. The statement 5Ø, "NEXT W," adds one to "W" and branches back to line 3Ø.
5. Our next area is calculated as "1Ø*11" or "11Ø."
6. The nested "FOR...NEXT" loop will display six areas with a length of 1Ø and a width of 1Ø, 11, 12, 13, 14, and 15.
7. When the "NEXT W" statement increments "W" to give it a value of "16," the loop ends and the next statement (6Ø NEXT L) is executed.
8. This results in the adding of one to "L" (so that it equals "11").
9. Statement 3Ø (FOR W = 1Ø TO 15) is encountered again, such that "W" is reset to "1Ø" and six more values are displayed with six widths and a length of 11.
10. This process continues until 24 more square footages are printed as the length increases 12, 13, 14, and 15 with six widths for each.
11. When the program terminates at line 7Ø, we see 36 values on the screen. "W" and "L" will each equal "16," as this was greater than the maximum value they were to achieve.

Suppose we need to increment the length or width by a value other than one? We could accomplish this by inserting the STEP clause. Modify line 3Ø to read as follows.

```
3Ø FOR W = 1Ø TO 15 STEP .5
```

RUN the program and see what happens. This time, twice as many areas will be printed for lengths of 1Ø through 15 as before. There will also be 12 widths for each as "W" moves from 1Ø to 1Ø.5 to 11 to 11.5, etc.

This will be more output than our screen can handle at one time. When the report code shows the screen to be full, use the CONT command (Continue) to display the rest of the output.

Now, test your command of tables, "FOR...NEXT" loops, and the VAL and STR$ functions by applying them to practical situations in the following Programmer's Check.

PROGRAMMER'S CHECK

2

Tables, Loops and Functions

Designing programs for such practical applications as sales tax calculations, temperature conversions, and numerical tables will use much of what you have learned thus far. If you have any difficulty understanding how to create appropriate loops or functions, check back to the text for help rather than simply guessing. However, thoughtful guesswork can be constructive, too, so you ought to follow your "strong hunches" now and then.

1. **This sales tax program requires you to consider the best way of setting up a design which allows the client to input various item costs and to obtain the total cost including tax.**

Program Name: IU9A1
Type: SALES TAX
Specifications:

Design and code a program which allows the user to enter up to 1ØØ different costs of various items, as in a shopping list. Print out each of the costs entered as soon as the user is done. Then, display the total amount of the items purchased, the sales tax (5%) and the total cost. Use the following test data:

INPUT

COSTS 1Ø.ØØ

15.75

6.5Ø

8.75

OUTPUT

2. **Design a program which will calculate Fahrenheit to Celsius conversions for a range of temperatures entered when the program is run. (HINT: Allow the user to INPUT the values used in the control variables "FOR...NEXT" loop.)**

3. **Write a program which loads up to 5Ø numbers into a table. Depending upon the length of the number, print each number so that all numbers align on the right as shown in the example below. (HINT: Use the STR$ and LEN and TAB functions.)**

EXAMPLE:

Column 1Ø
1ØØ
1Ø
1825
6

(Answers on Pages 20-22)

MORE ABOUT TABLES

CHARACTER (STRING) ARRAYS

String data may also be stored into tables. Unlike numeric tables, string arrays need not be defined or dimensioned. The subscript merely references the characters to be accessed from the string variable. For example, the following statements should display the letter "C" when run.

```
1Ø LET N$ = "ABCDEF"

2Ø PRINT N$(3)

3Ø STOP
```

The literal subscript extracts characters from the variable, "N$". (We can also use a variable subscript as before, too.) In this case, the third character is extracted ("N$(3)" is equal to "C").

If we want to see the first three characters from "N$," we can indicate the first and last letters within parentheses, connected by the word "TO" as follows:

```
2Ø PRINT N$(1 TO 3)
```

What would be the result? Why, go ahead and try it. The result is "ABC" appearing on the screen.

SUBSTRINGING

Substringing (or slicing) refers to the ability to break a string value into pieces. When applied to simple string variables (that is, strings which are *not* tables), parentheses are used to reference particular characters within the string.

Say, for example, we were to assign the value "AX3B" to the variable "Z$" as in the command:

```
1Ø LET Z$ = "AX3B"
```

If we then wished to display only the first character in this string, we could enter the statement:

```
2Ø PRINT Z$(1)
```

Try it. You will see the value "A" displayed on the screen. Similarly, "Z$(2)" is "X"; "Z$(3)" is "3"; and "Z$(4)" is "B".

Change Line 2Ø to one of these values and RUN the program for proof of this slicing ability.

We can also access several consecutive characters within a string by entering the first and last characters to be referenced in the parentheses. Let's take another example to demonstrate this feature. If we let A$ equal "JKLMNO", then we could print the first three characters with a print command.

```
1Ø LET A$ = "JKLMNO"

2Ø PRINT A$(1 TO 3)
```

This statement will display the first character of "A$" *through* the third or "JKL." Likewise,

```
PRINT A$(5 TO 6)
```

would display "NO", the fifth and sixth characters.

It is also possible to omit the beginning or ending characters when we want to start from the first character in the string or when we wish to include the remaining positions, respectively. For example,

```
PRINT A$(   TO 4)
```

would assume to begin at position "1" and would display "JKLM". By leaving out the end position, as in

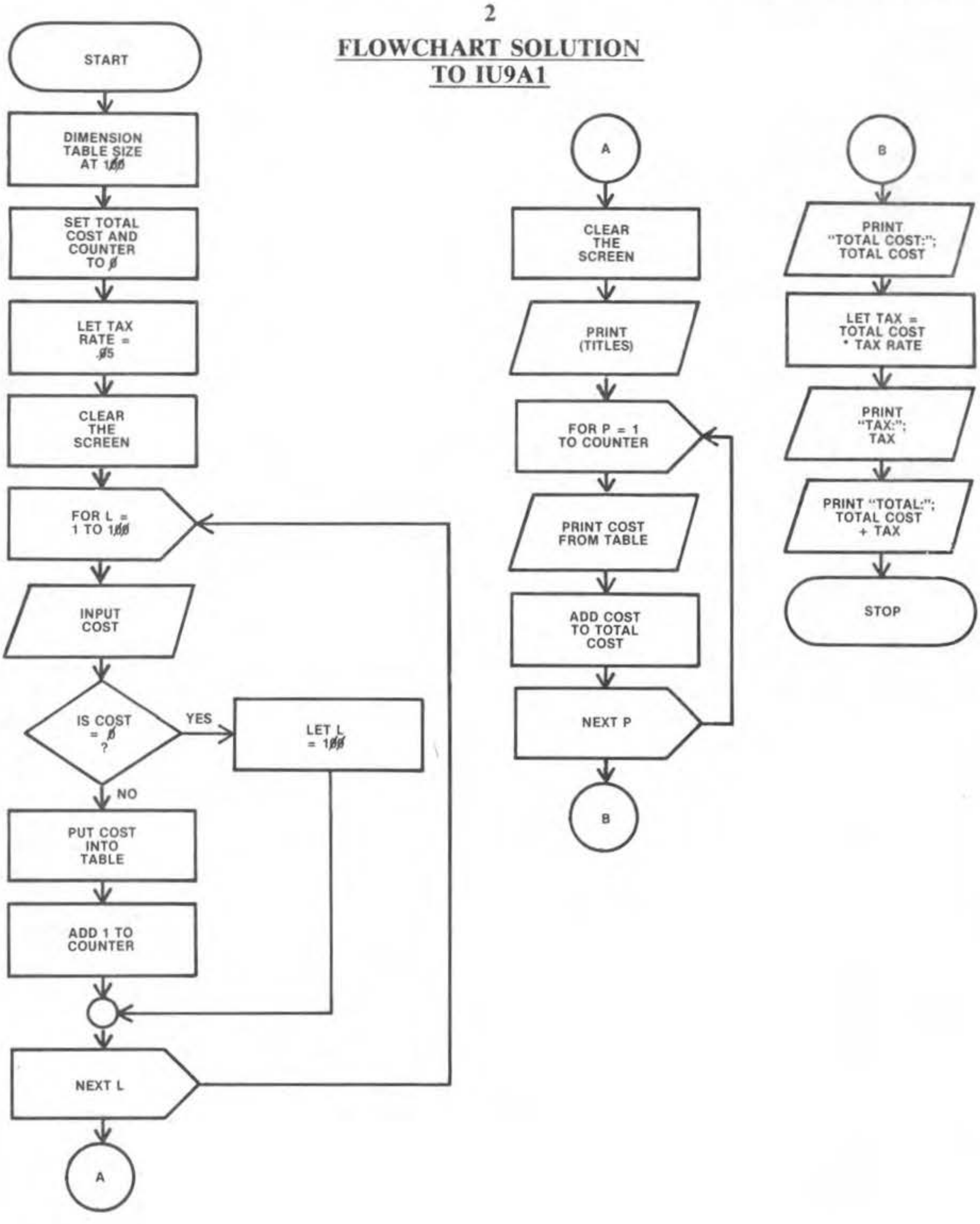
```
PRINT A$(3 TO   )
```

we would start at the third character and continue for the entire length of the string, or "LMNO".

PROGRAMMER'S CHECK ANSWERS

2

FLOWCHART SOLUTION
TO IU9A1



Programmer's Check 2 Answer (continued)

Program Solution to IU9A1

```
1Ø REM IU9A1 SALES TAX
2Ø REM I....COST OF AN ITEM
3Ø REM R....TAX RATE — 5 PCT.
4Ø REM T1....TOTAL COST
5Ø REM T2....TOTAL ITEMS ENTERED
6Ø REM C....COST TABLE
7Ø REM L....CONTROL VARIABLE —
   LOAD
8Ø REM P....CONTROL VARIABLE —
   PRINT
9Ø REM X....TOTAL TAX
1ØØ DIM C(1ØØ)
11Ø LET T1 = Ø
12Ø LET T2 = Ø
13Ø LET R = .Ø5
14Ø CLS
145 PRINT AT 21,Ø; "ENTER COSTS"
15Ø FOR L = 1 TO 1ØØ
16Ø INPUT I
17Ø IF I = Ø THEN GOTO 21Ø
18Ø LET C(L) = I
19Ø LET T2 = T2 + 1
2ØØ GOTO 22Ø
21Ø LET L = 1ØØ
22Ø NEXT L
23Ø CLS
24Ø PRINT TAB 8; "SHOPPING LIST"
25Ø PRINT "COSTS"
26Ø FOR P = 1 TO T2
27Ø PRINT C(P)
28Ø LET T1 = T1 + C(P)
29Ø NEXT P
3ØØ PRINT "TOTAL COST: "; TAB 12; T1
31Ø LET X = T1*R
32Ø PRINT "TAX:"; TAB 12; X
33Ø PRINT "TOTAL:"; TAB 12; T1 + X
34Ø STOP
```

Program Solution to Number 2

```
1Ø INPUT A
2Ø INPUT B
3Ø CLS
4Ø PRINT "FAHRENHEIT"; "CELSIUS"
5Ø FOR F = A TO B
6Ø LET C = (F — 32)*(5/9)
7Ø PRINT F,C
8Ø NEXT F
9Ø STOP
```

(continued)

Programmer's Check 2 Answer (continued)

Program Solution to Number 3

```
10 DIM T(50)
15 LET T1 = 0
20 FOR L = 1 TO 50
30 INPUT N
40 IF N = 0 THEN GOTO 80
50 LET T(L) = N
60 LET T1 = T1 + 1
70 GOTO 90
80 LET L = 50
90 NEXT L
100 FOR P = 1 TO T1
110 LET R$ = STR$ T(P)
120 LET R = LEN R$
130 PRINT TAB 11 - R; T(P)
140 NEXT P
150 STOP
```

CONCATENATION

Concatenation is the process of combining string values into a single value (the opposite of substringing). When applied to strings, the plus sign (+) does not mean an arithmetic addition, but concatenation. Take the following example:

```
10 LET A$ = "BA"
20 LET B$ = "SIC"
30 LET C$ = A$ + B$
40 PRINT C$
```

What do you think would be displayed? If you guessed "BASIC", then you are absolutely correct! Statement 30 put the values of "A$"(BA) and "B$"(SIC) into a single variable, "C$". This process could just as well be used with literals. Say that we had two variables, "F$" and "L$". We could write a BASIC program which would prompt the user to enter a first name for "F$" and a last name for "L$".

We could then combine these into a single variable, "N$" as in:

```
10 INPUT F$
20 INPUT L$
30 LET N$ = F$ + L$
```

The only problem is that if we were to print "N$", we would find no space between the two parts of the name. We could correct this by modifying line 30 to read:

```
30 LET N$ = F$ + "   " + L$
```

Then "N$" would contain a space between each part of the name.

Thus far, we have limited this discussion to the manipulation of string values. But what if we had to deal with numeric variables? We could use the "STR$" function to convert a numeric into a string. Then we would be able to utilize substringing and concatenation on the converted variable.

Finally, we could use the VAL function to change it back into a number!

This process would be useful if, for example, we wanted to compare dates.

If we had two dates in a month-month, day-day, year-year format (MMDDYY), they could look like this:

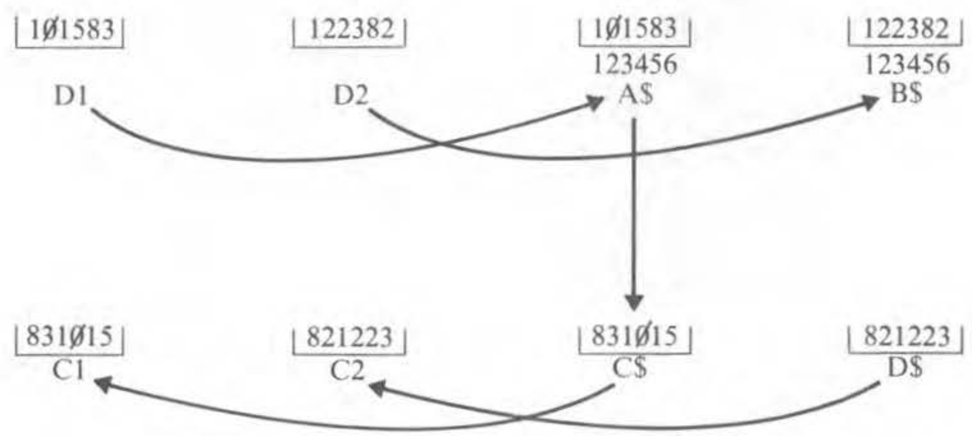
FIELD	VARIABLE NAME	CONTENTS
FIRST DATE	D1	101583
SECOND DATE	D2	122382

Our two dates have values which correspond to October 15, 1983, and December 23, 1982. Comparing these dates, as is, would produce some strange results—that is, the second date is considered to be *less than* the first. This is because the computer views these two values as the numbers 1Ø1,583 and 122,382 respectively. Actually, in a date, the year's digits must be considered to be the *most significant values*. What we must do is to convert the dates into the YYMMDD format; then, our values would be able to be compared. See if you can follow this program to understand how it works:

```
1Ø LET D1 = 1Ø1583

2Ø LET D2 = 122382

3Ø LET A$ = STR$ D1

4Ø LET B$ = STR$ D2

5Ø LET C$ = A$(5 TO 6) +
   A$(1 TO 4)

6Ø LET D$ = B$(5 TO 6) +
   B$(1 TO 4)

7Ø LET C1 = VAL C$

8Ø LET C2 = VAL D$

9Ø PRINT D1; " IS LESS THAN "; D2

1ØØ PRINT " BUT, "; C1; " IS
    GREATER THAN "; C2

11Ø STOP
```

On lines 1Ø and 2Ø the numeric values of the dates in the MMDDYY format are assigned to variables D1 and D2. Then, statements 3Ø and 4Ø convert these dates into string variables, using the STR$ function. Once converted, lines 5Ø and 6Ø are used to slice and concatenate the dates, using the last two digits (the years) in A$ and B$ as the first two digits of C$ and D$. The four places of the month and day are then appended to the rightmost positions of C$ and D$.

The next two instructions (lines 7Ø and 8Ø) change these new string variables back into numerics, where they can then be used in a numeric comparison. This short program "example," however, merely prints out the values—both old and new.



SEARCHING STRING ARRAYS

We could use the instructions we have just used to "scan" an array in order to display the number of times a particular character appears in a string. We could, for instance, scan a number in order to see how many zeros were contained in it. Such a program would look like this:

```
1Ø INPUT N
2Ø LET N$ = STR$ N
3Ø LET L = LEN N$
4Ø LET C = Ø
5Ø FOR S = 1 TO L
6Ø IF N$(S) = "Ø" THEN LET C =
   C + 1
7Ø NEXT S
8Ø PRINT AT 12,Ø; "THERE ARE "; C;
   " ZEROS IN "; N$
9Ø STOP
```

In this program, the first instruction prompts us to enter a number. Let's enter the number "5ØØ56." The next two statements convert this to a string variable (N$) and determine the length (L). In this case, "L" will be "5."

The variable "C" will count the number of zeros we encounter as the string is scanned; it must, however, be initialized to zero.

A "FOR. . .NEXT" loop sets the variable "S" to "1" (line 5Ø). The "IF" statement then tests to see whether the first character is a zero. If it is a zero, one is added to "C."

Statement 7Ø increments the control variable and subscript "S" by one and continues to scan until all five (L) characters are checked. The PRINT statement should then display "2," as there are two zeros in 5ØØ56.

RUN the program and watch it work. Did you leave spaces in Line 8Ø so the words and numbers are separated on the screen? Line 8Ø should read as follows:

ENTER different values, noting that ØØ9 would show no zeros and 1ØØ.ØØ will show "2". Leading zeros and zeros after the decimal point will not be counted, as they have no affect upon the value of a number and, therefore, are not maintained.

We could modify this program to scan for the presence of any character we would like to check. We could also change the character, if we like. Try this on your own and see if you can scan characters.

```
1Ø INPUT N$
2Ø INPUT C$
3Ø INPUT R$
4Ø LET L = LEN N$
5Ø FOR S = 1 TO L
6Ø IF N$(S) = C$ THEN LET N$(S) = R$
7Ø NEXT S
8Ø PRINT N$
9Ø STOP
```

The "N$" is the character data to be scanned. Say we enter "BROWN, JOE". "C$" is the character we will be looking for in this case. We will ENTER a comma (,) here. When the next prompt is displayed, ENTER a blank space. The program should replace the comma with a space. RUN the program. Can you see how it works? Try entering a blank space for each of the two prompts. Is that a better display?

PROGRAMMER'S REMINDER

Word processors make use of the "scan" and "scan and replace" techniques. Word processors are actually just computers and/or software dedicated to manipulating text data.

SEARCHING NUMERIC TABLES

One of the greatest uses of tables is being able to store and maintain data independently of our main input records. For example, we can maintain a table for bonuses given by a company to employees based upon codes. The table might look like this:

BONUS TABLE	
CODE	AMOUNT
1	$15.ØØ
2	$2Ø.ØØ
3	$3Ø.ØØ

FIGURE 8—How remote are these conservationists and deer from the high technology world of the computer? You might be surprised to know that the two worlds exist side by side. In the Canadian wilderness, certain fawns are monitored each year to track migrations and living patterns. Data is maintained on computers so that scientists, wildlife managers and naturalists may learn more about these and other animals.

This bonus table could be loaded into main storage initially, but could be "saved" (onto cassette tape), along with the program. Our program can then access that table without the need to reenter the table data.

Imagine that we have input records consisting of an employee number, hours worked and a bonus code. Our output will require an employee name, regular pay, bonus pay and total pay. The bonus pay will be determined by directly accessing the bonus pay table with the bonus pay code on input. But where will the employee name and regular pay come from?

We will need to establish some other tables within our program. One will contain a list of employee numbers. Another will contain employee names. A third will have the hourly pay rate each employee is to receive. Our three additional tables will look like this:

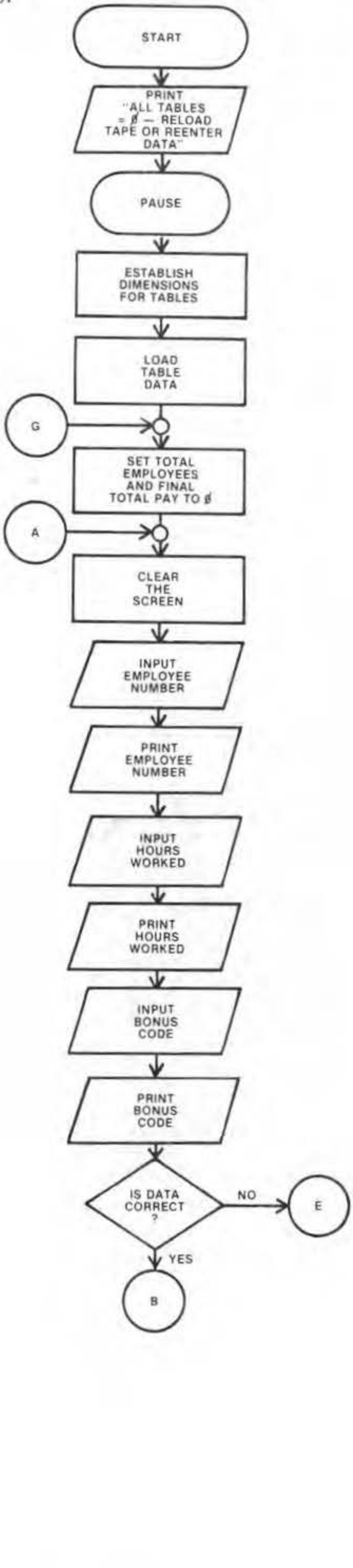
EMPLOYEE MASTER FILE TABLES		
EMPLOYEE NUMBER	EMPLOYEE NAME	PAY RATE
1Ø1	STEVE STONE	6.75
285	BILL WILLIAMS	8.ØØ
46Ø	JUDY JOHNSON	7.5Ø
819	ANN BROWN	8.25

Notice the way in which the data is arranged. Although we have three separate tables, they are aligned in such a way that the first employee number corresponds to the first employee name and the first pay rate. This data can be entered and maintained apart from the main logic of our program.

The data in the tables is relatively permanent. It will be changed much less frequently than the program, which will be "run" on a weekly basis. We can refer to this data as our "master file".

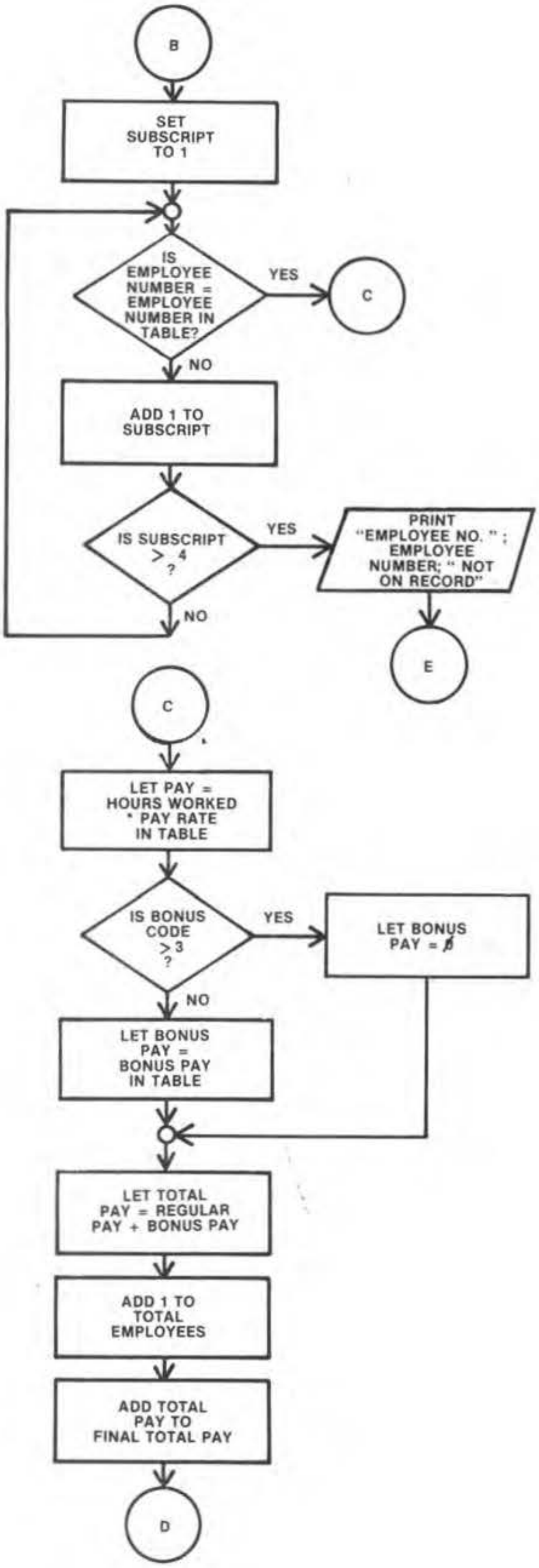
When the weekly payroll program is run, the operator will enter the employee number, hours worked for the week and bonus code. This data composes the "transaction file".

A flowchart solution to this problem is as follows:

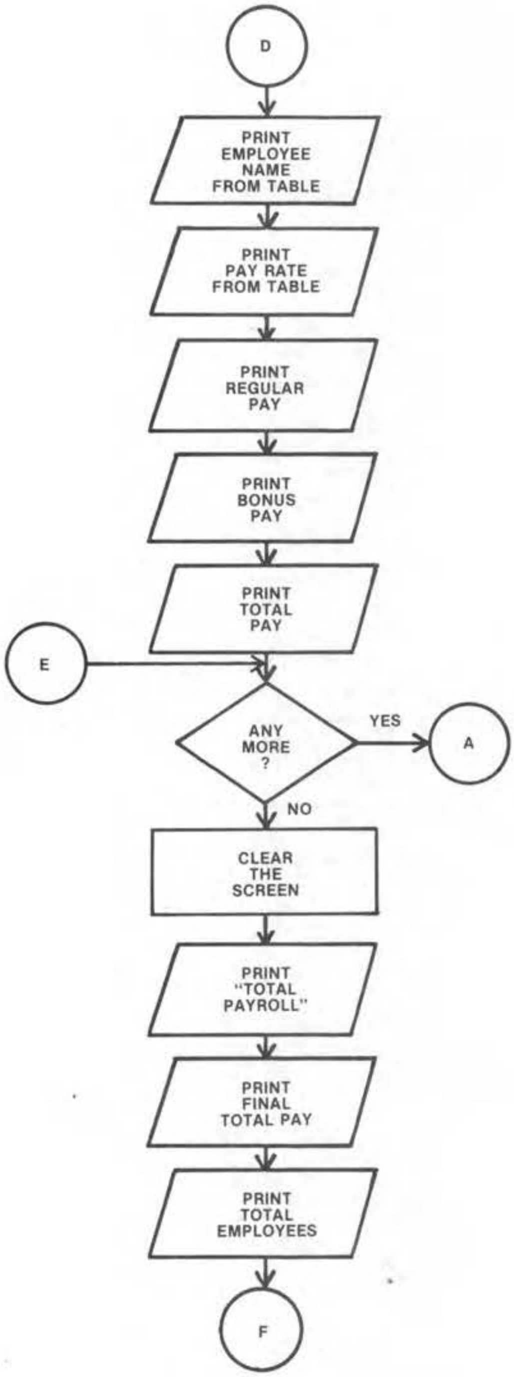


Below is the program that emerges from the design.

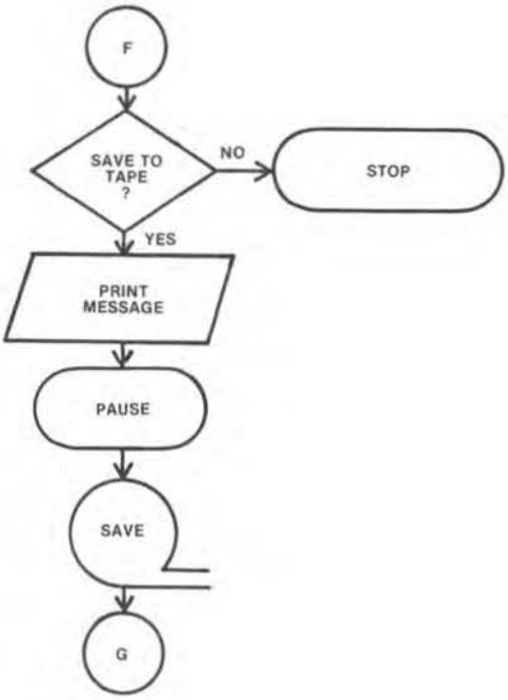
```
1Ø REM WEEKLY PAYROLL
   PROGRAM
2Ø REM MASTER FILE:
3Ø REM E(  ) EMPLOYEE NUMBER
   TABLE
4Ø REM N$(  ) EMPLOYEE NAME
   TABLE
5Ø REM R(  ) PAY RATE TABLE
6Ø REM B(  ) BONUS CODE TABLE —
   NOT PART OF MASTER FILE
62 REM SUBSCRIPTS:
64 REM L SUBSCRIPT FOR MASTER
   TABLES
66 REM M SUBSCRIPT FOR BONUS
   TABLE
68 REM S SUBSCRIPT TO SEARCH
   MASTER TABLES
7Ø REM ACCUMULATORS:
8Ø REM T1 TOTAL EMPLOYEES
   PROCESSED
9Ø REM T2 FINAL TOTAL PAY
1ØØ REM TRANSACTION FILE:
11Ø REM N EMPLOYEE NUMBER
12Ø REM H HOURS WORKED
13Ø REM C BONUS CODE
132 REM OUTPUT:
134 REM P1 REGULAR PAY
136 REM P2 BONUS PAY
138 REM P3 TOTAL PAY
14Ø REM A$ RESPONSE TO ANY
    MORE?
15Ø PRINT "ALL TABLES = Ø —
    RELOAD TAPE OR REENTER
    DATA"
155 PAUSE 32767
16Ø DIM E(4)
17Ø DIM N$(4,15)
18Ø DIM R(4)
19Ø DIM B(3)
2ØØ FOR L = 1 TO 4
21Ø PRINT AT 21,Ø; "ENTER
    EMPLOYEE NO. "
22Ø INPUT E(L)
23Ø PRINT AT 21,Ø; "ENTER
    EMPLOYEE NAME"
24Ø INPUT N$(L)
25Ø PRINT AT 21,Ø; "ENTER PAY
    RATE      "
26Ø INPUT R(L)
27Ø NEXT L
28Ø FOR M = 1 TO 3
29Ø PRINT AT 21,Ø; "ENTER BONUS
    PAY"
3ØØ INPUT B(M)
31Ø NEXT M
32Ø REM WEEKLY RUN BEGINS HERE
33Ø LET T1 = Ø
34Ø LET T2 = Ø
35Ø CLS
36Ø PRINT AT 1,Ø; "EMPLOYEE NO."
37Ø INPUT N
38Ø PRINT AT 1,14; N
39Ø PRINT AT 2,Ø; "HOURS
    WORKED"
4ØØ INPUT H
41Ø PRINT AT 2,14; H
42Ø PRINT AT 3,Ø; "BONUS CODE"
43Ø INPUT C
44Ø PRINT AT 3,14; C
45Ø PRINT AT 21,Ø; "IS ALL DATA
    CORRECT (Y/N)?"
46Ø INPUT A$
47Ø IF A$ = "N" THEN GOTO 67Ø
475 PRINT AT 21,Ø; "
    "
```




```
48Ø REM VALID DATA — SEARCH
    TABLES
49Ø LET S = 1
5ØØ IF N = E(S) THEN GOTO 56Ø
51Ø LET S = S + 1
52Ø IF NOT S > 4 THEN GOTO 5ØØ
53Ø PRINT AT 6,Ø; "EMPLOYEE
    NO. " ; N; " NOT ON RECORD"
54Ø GOTO 67Ø
55Ø REM EMPLOYEE ON RECORD —
    DETERMINE PAY
56Ø LET P1 = H*R(S)
57Ø IF C > 3 THEN LET P2 = Ø
58Ø IF NOT C > 3 THEN LET P2 =
    B(C)
59Ø LET P3 = P1 + P2
6ØØ LET T1 = T1 + 1
61Ø LET T2 = T2 + P3
```



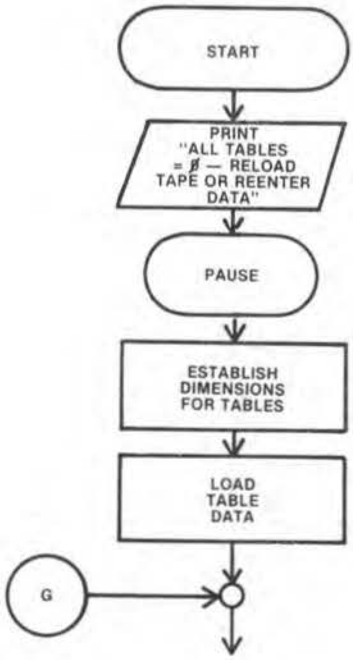

```
62Ø PRINT AT 6,Ø; "EMPLOYEE
    NAME"; TAB 14; N$(S)
63Ø PRINT AT 8,Ø; "PAY RATE"; TAB
    14; R(S)
64Ø PRINT AT 1Ø,Ø; "REGULAR PAY";
    TAB 14; P1
65Ø PRINT AT 11,Ø; "BONUS PAY";
    TAB 14; P2
66Ø PRINT AT 12,Ø; "TOTAL PAY";
    TAB 14; P3
67Ø PRINT AT 21,Ø; "ANY MORE
    (Y/N)?"
68Ø INPUT A$
69Ø IF A$ = "Y" THEN GOTO 35Ø
7ØØ CLS
71Ø PRINT TAB 4; "TOTAL
    PAYROLL"
72Ø PRINT AT 5,Ø; "TOTAL PAY ";
    TAB 22; T2
73Ø PRINT AT 8,Ø; "TOTAL
    EMPLOYEES "; TAB 22; T1
```




```
74Ø PRINT AT 21,Ø; "SAVE TO TAPE
    (Y/N)?"

75Ø INPUT A$

76Ø IF A$ = "N" THEN STOP

77Ø PRINT AT 21,Ø; "SET TAPE TO
    RECORD — PRESS ANY KEY"

775 PAUSE 32767

78Ø SAVE "PAYROLL"

79Ø GOTO 33Ø
```

Before you enter, run and save this program, let's examine the logic, step-by-step. First, the master file and bonus tables are established.

LOGIC FOR MASTER FILE AND BONUS TABLES




```
15Ø PRINT "ALL TABLES = Ø —
    RELOAD TAPE OR REENTER
    DATA"

155 PAUSE 32767

16Ø DIM E(4)

17Ø DIM N$(4,15)

18Ø DIM R(4)

19Ø DIM B(3)

2ØØ FOR L = 1 TO 4

21Ø PRINT AT 21,Ø; "ENTER
    EMPLOYEE NO. "

22Ø INPUT E(L)
```

```
23Ø PRINT AT 21,Ø; "ENTER
    EMPLOYEE NAME"

24Ø INPUT N$(L)

25Ø PRINT AT 21,Ø; "ENTER PAY
    RATE      "

26Ø INPUT R(L)

27Ø NEXT L

28Ø FOR M = 1 TO 3

29Ø PRINT AT 21,Ø; "ENTER BONUS
    PAY"

3ØØ INPUT B(M)

31Ø NEXT M
```

What's the "PRINT" statement at line 15Ø doing at the beginning of our program? It's a message informing the user that if the program is executed via a RUN command, all the master file table data will be reinitialized to zero. *In other words, the file table data will be lost.*

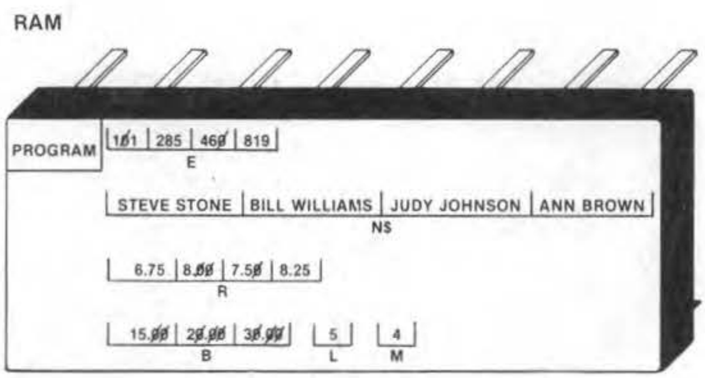
Once the table data has been entered the first time, it will be saved with the program onto magnetic tape. We won't have to reload the table; instead, we can skip to the logic which will produce the weekly payroll report.

The PAUSE statement on line 155 will allow the user to BREAK the program and reload from the tape. However, since the data has to be entered the first time, pressing any key allows the program to continue.

The DIM statements on lines 16Ø through 19Ø establish the dimensions of the four tables. Note that the employee name table sets up enough storage for 15 characters per name.

Immediately following are two "FOR... NEXT" loops which issue prompts and load data. The first loads each of the four employee numbers, names and pay rates into the master file tables. The second loop loads the amount for the three bonus codes.

When these statements have been executed, and we have loaded all our table data, this is how main storage will look.



Notice what happens in main storage. Although we have three separate master file tables (E,R and N$), they were entered so that the first element of each record is the first employee (1Ø1, STEVE STONE, 6.75); the second elements correspond to the second employee, and so on.

Notice, too, the bonus code table. It is established with three bonus pays for the three bonus codes.

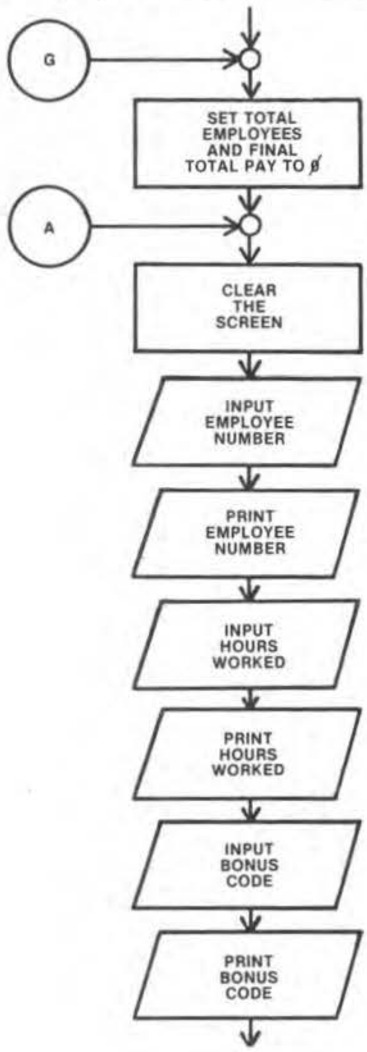
The control subscripts (L and M) now have values exceeding the upper limits of their "FOR. . .NEXT" loops (5 and 4 respectively).

We could have written the logic so that all four employee numbers would be entered, then all four names, and finally all four pay rates. This would work as well! The important thing is that the master records "line up."

As long as the DIM statements are not again executed via a GOTO or RUN command, they will remain resident in main storage. These instructions (16Ø through 31Ø) could even be deleted from the program, at this point, so even a GOTO command wouldn't wipe out the data (but a RUN command still would!).

The only problem with deleting these statements is that it would then be difficult to reenter the table data, if it had to be changed some time in the future. Let's examine the payroll run logic.

PAYROLL RUN LOGIC




```
33Ø LET T1 = Ø
34Ø LET T2 = Ø
35Ø CLS
36Ø PRINT AT 1,Ø; "EMPLOYEE NO."
37Ø INPUT N
38Ø PRINT AT 1,14; N
39Ø PRINT AT 2,Ø; "HOURS
    WORKED"
4ØØ INPUT H
41Ø PRINT AT 2,14; H
42Ø PRINT AT 3,Ø; "BONUS CODE"
43Ø INPUT C
44Ø PRINT AT 3,14; C
```

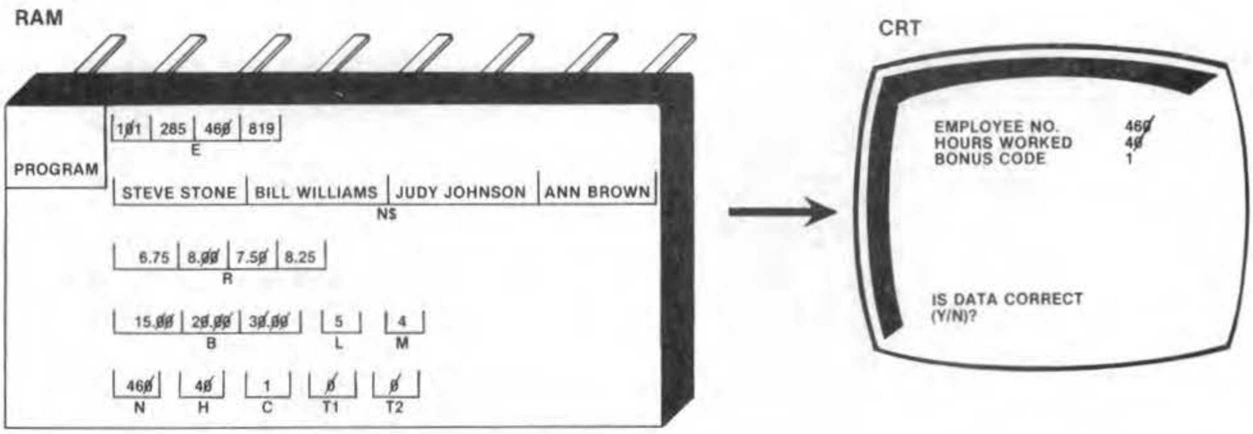
Every week the program is executed, the accumulators will be set to zero. The screen is then cleared and prompts are issued for each input transaction record.

Notice that in this program, the prompts are printed near the top of the screen. After data has been submitted as input, the data is displayed beside the prompt. This allows the user to "see" what has been entered.

Our next set of logic will give the user the opportunity to "cancel" this transaction, should it appear incorrect. Here is an example of a transaction entered.

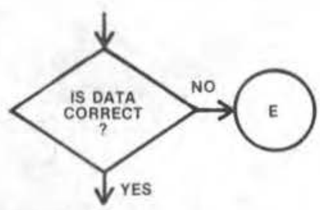
EMP. NO.	HOURS WORKED	BONUS CODE
46Ø	4Ø	1

Before this record is processed, main storage will look like this:



The logic used to cancel data just entered is called "skip over this record" logic. Here is what it looks like as design flow and program elements.

"SKIP OVER THIS RECORD" LOGIC



45Ø PRINT AT 21,Ø; "IS ALL DATA CORRECT (Y/N)?"

46Ø INPUT A$

47Ø IF A$ = "N" THEN GOTO 67Ø

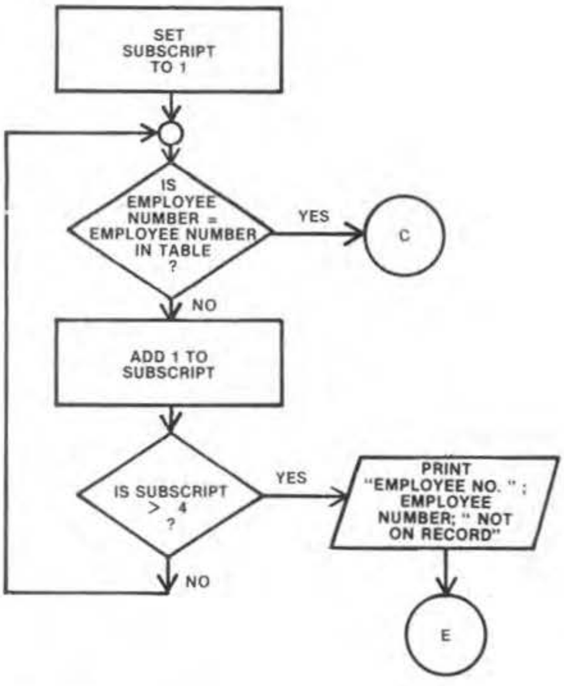
475 PRINT AT 21,Ø; "   "

From the user's standpoint, a message is displayed at the bottom of the screen. If the user is happy with the data just entered, a response of "Y" can be entered and the message will disappear. (Line 475 prints blank spaces over the message.)

If the user is unhappy with the data, then a response of "N" is entered. The "N" causes a branch to occur to the end of the loop. Then, the computer will ask whether processing should be continued for another employee.

In this example, the data has been entered properly and the user responds with a "Y." Now, detail processing takes place as shown in the flowchart and program.

DETAIL PROCESSING LOGIC




```
49Ø LET S = 1

5ØØ IF N = E(S) THEN GOTO 56Ø

51Ø LET S = S + 1

52Ø IF NOT S > 4 THEN GOTO 5ØØ

53Ø PRINT AT 6,Ø; "EMPLOYEE
    NO. " ; N; " NOT ON RECORD"

54Ø GOTO 67Ø
```

This portion of our logic searches the employee number table (E) in our master file to determine which element matches the employee number entered (N).

It is assumed that the employee we are looking for is first in the tables. This logic is accomplished by setting a subscript (S) to "1". Then, we check to see if the assumption is correct. Is the employee number just entered, "N", the same as the first employee number in the table (E(L))?

In this case, however, there is not a match. So, "1" is added to the subscript, and we continue the search with the second element of the table. Before looping back, we need to test to see if all elements have already been examined. We shouldn't assume that the user either

1. correctly entered the data, or
2. knows which employee numbers are in the table.

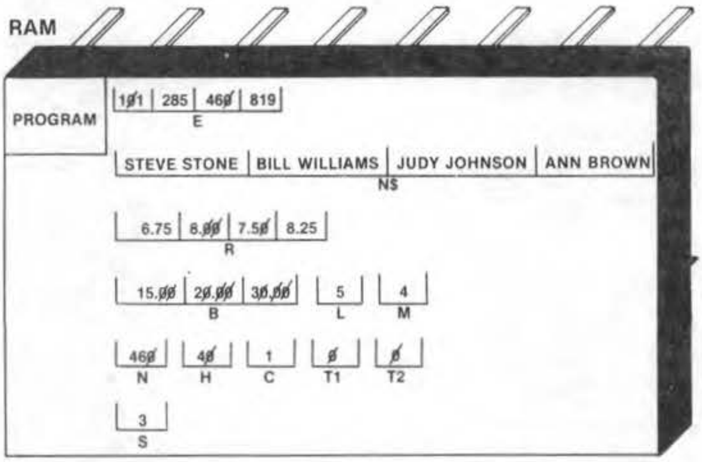
If all elements had been checked previously, an "error message" is printed on the screen. The error message indicates that this employee number is not on the master file and, therefore, cannot be correctly processed.

Since we have yet to check the second, third and fourth elements, we branch back to check again. And once more, we will not find a match on the second try. When "S" equals "2", it is not a match. "S" then becomes "3" and we loop again.

FIGURE 9—Computers have revolutionized the printing industry. Entire pages of newspapers can be created on the computer-driven typesetting equipment. And by using "communication software" and modem, authors using microcomputers can send manuscripts to publishers via telephone.

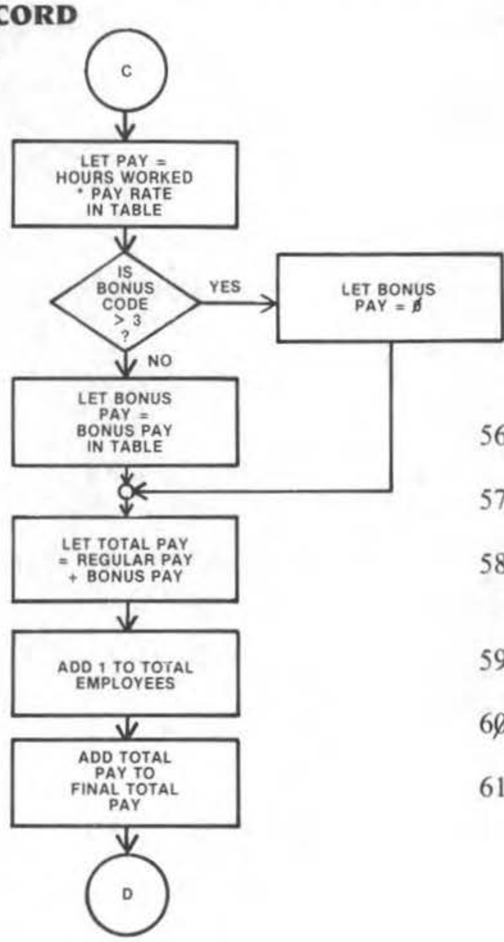
This time, we do have a match. "N", which equals 46Ø, is equal to "E(S)" when "S" equals "3". Now that we have found the employee we were looking for, we can exit the loop.

Notice in the main storage illustration that the search subscript "S" is left at the value "3". This allows us to access the third element from each of the other two master file tables.



Having matched the employee number entered with that found in the master file, we can see what happens after it is located.

PROCESS THE RECORD IS FOUND IN THE MASTER FILE




```
56Ø LET P1 = H*R(S)
57Ø IF C > 3 THEN LET P2 = Ø
58Ø IF NOT C > 3 THEN LET P2 =
    B(C)
59Ø LET P3 = P1 + P2
6ØØ LET T1 = T1 + 1
61Ø LET T2 = T2 + P3
```

At this point, having found a "match", we can multiply the hours worked (H) by the third pay rate in the table, "R(S)", to determine the regular pay. This record will compute the pay as 4ø times 7.5ø or $3øø.

In the logic just performed, we have been using the "alternating format" of tables. In the alternating format, a search argument (N) is compared for a match in the argument table (E). The functions (N$ and R) are then accessed.

However, this type of logic is not used in determining the bonus pay. Instead, the "direct access" method is employed. Since our valid bonus codes are one, two and three, the bonus code entered can be used as the subscript for the bonus pay table.

PROGRAMMER'S REMINDER ABOUT "DIRECT ACCESS"

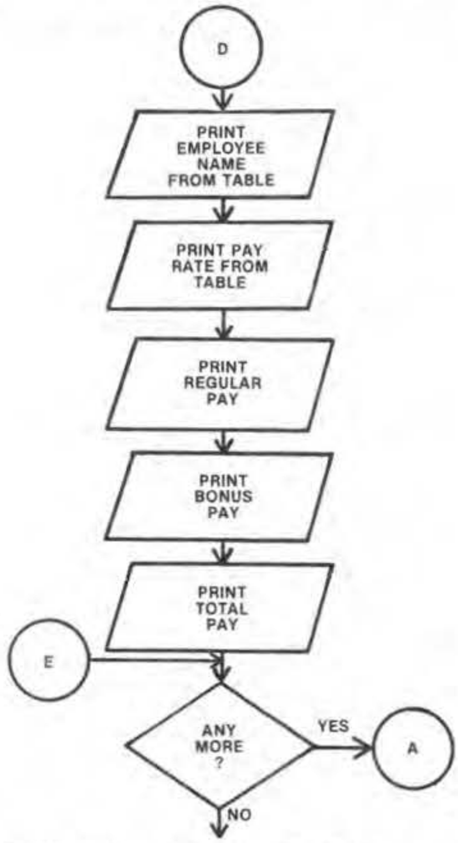
Direct access is only used if the search argument corresponds with the position of the element in the table. This could be used for the employee number table if the employee number 1ø1 was the 1ø1st element of the table, number 285 was the 285th element of the table, etc. But this would have necessitated the dimensioning of huge tables with the 1st through the 1øøth elements left blank. Direct access to tables is, therefore, limited in its use.

Before we acquire the correct bonus pay, our logic checks to see whether the code entered can be used as the subscript. In other words, does the code equal one, two or three? If the code is greater than three, we'll assume that the bonus pay should be zero. If the code is not greater than three, then the bonus table entry, according to the value of the code, will be assigned to the bonus pay, "P2".

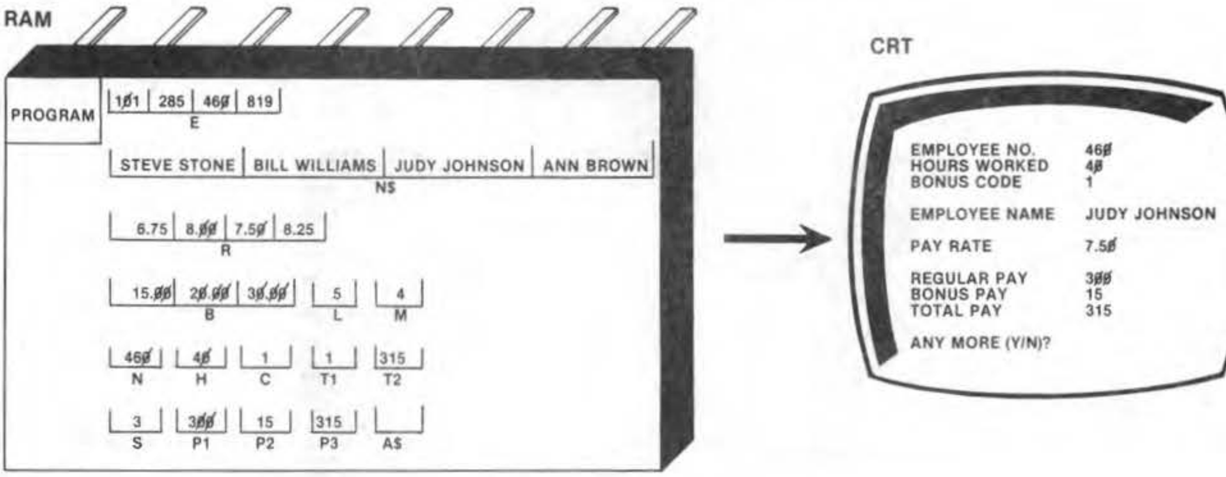
The next three calculations determine the total pay by adding the regular pay to the bonus pay and then adding to our accumulators for the final totals.

THE OUTPUT DISPLAY LOGIC

The processing of this record then terminates with the output display logic as shown here.




```
62ø PRINT AT 6,ø; "EMPLOYEE
     NAME"; TAB 14; N$(S)
63ø PRINT AT 8,ø; "PAY RATE"; TAB
     14; R(S)
64ø PRINT AT 1ø,ø; "REGULAR PAY";
     TAB 14; P1
65ø PRINT AT 11,ø; "BONUS PAY";
     TAB 14; P2
66ø PRINT AT 12,ø; "TOTAL PAY";
     TAB 14; P3
67ø PRINT AT 21,ø; "ANY MORE
     (Y/N)?"
68ø INPUT A$
69ø IF A$ = "Y" THEN GOTO 35ø
```



At this point, we'll assume that "Y" was entered for "A$" and our logic then loops back to process another employee.

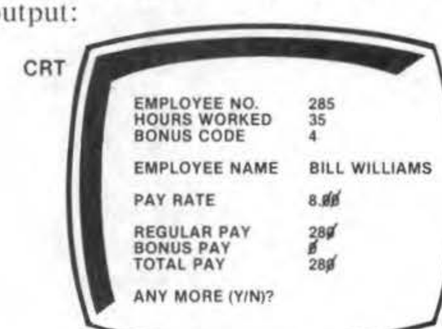
FIGURE 10—This poor woman never will get the upper hand on her paper work—unless she gets help from a programmer. Most paper work is repetitive, using the same formats, information and data over and over again. Correspondence, payroll, billing—you name it—can be processed using loops, dimensions, and arrays.

SECOND RECORD PROCESSING LOGIC

Let's process the second record as:

EMP. NO.	HOURS WORKED	BONUS CODE
285	35	4

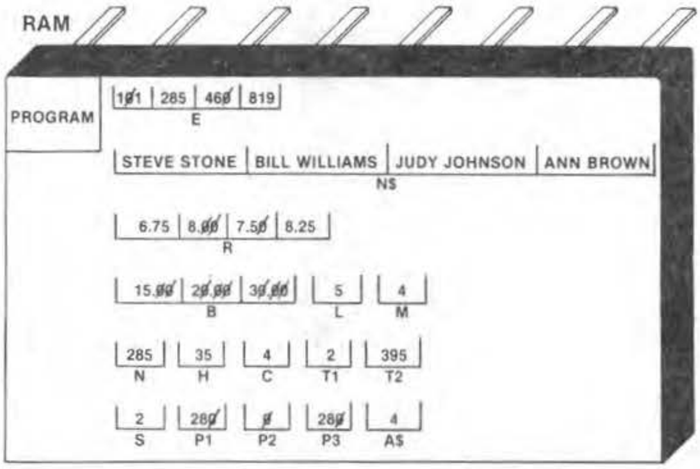
See if you can walk through the logic for this record on your own. This will be the output:



Is this what you expected? After being entered and verified, this record will find a match between the search argument (285) and the second element in the argument table. The functions give us the name of "BILL WILLIAMS" and a pay rate of "8.ØØ".

Notice that the bonus pay is zero. This is because the bonus code entered was greater than three. Therefore, it could not be used as a valid subscript for the bonus pay table.

Here is how main storage appears as we prepare to enter our third input record:

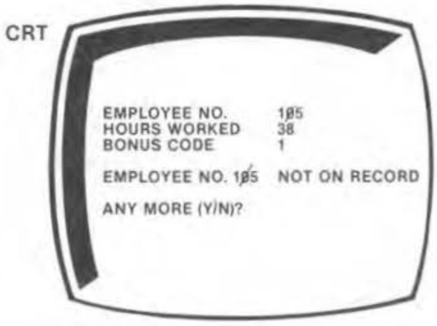


THIRD RECORD PROCESSING LOGIC

To see if our program *really* works, let's see how it handles an invalid record. Try this one:

EMP. NO	HOURS WORKED	BONUS CODE
1Ø5	38	1

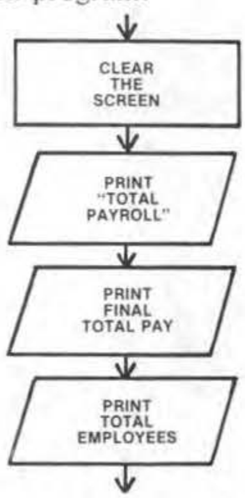
On this pass, even when the operator chooses to accept this invalid employee number, our program searches through the entire argument table without ever finding a match. When, on line 52Ø the search subscript "S" equals "4" (indicating that all master file employee numbers have been checked), an error message is displayed. The output display would appear as follows:



The internal memory will be little changed from before, with the exceptions of our input variables (N, H and C) and the search subscript "S," which is left at "4."

FINAL TOTAL LOGIC

When "N" is entered for "A$", here is how the final total logic would appear in flowchart and program.

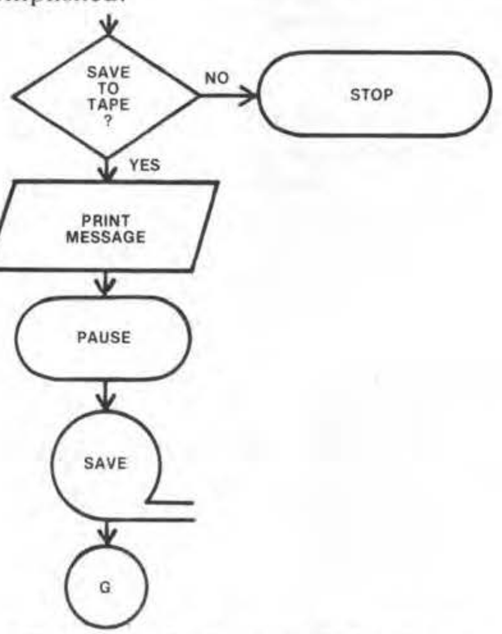



```
7ØØ CLS
71Ø PRINT TAB 4; "TOTAL PAYROLL"
72Ø PRINT AT 5,Ø; "TOTAL PAY ";
    TAB 22; T2
73Ø PRINT AT 8,Ø; "TOTAL
    EMPLOYEES "; TAB 22; T1
```

These statements do not affect any of our program's variables. They produce this output:

"SAVING THE DATA" LOGIC

Rather than terminating the program, the last few processing steps in this logic allow the user to save the program—along with the variables—onto cassette tape. Here is how this is accomplished.




```
74Ø PRINT AT 21,Ø; "SAVE TO TAPE
    (Y/N)?"
75Ø INPUT A$
76Ø IF A$ = "N" THEN STOP
77Ø PRINT AT 21,Ø; "SET TAPE TO
    RECORD — PRESS ANY KEY"
775 PAUSE 32767
78Ø SAVE "PAYROLL"
79Ø GOTO 33Ø
```

Line 74Ø requests a "Y" response, if the user wants the program and table data saved onto tape. If "N" is given, then the program terminates.

When a "Y" is given, another message appears on the screen. *The computer can only save the program when the tape cassette player is on-line (attached). The cassette tape must also be properly positioned and ready for recording. The PLAY and RECORD buttons must be pushed.*

FIGURE 11—What does your future hold within a given occupation? Nearly every job category has been programmed on computer and data is constantly being collected. Projections based on computer data can give you a perspective on future earnings, job placement and advancement.

PROGRAMMER'S REMINDERS ABOUT USING "MAGNETIC TAPE"

You are about to experience one of the great joys of computer programming —saving your creation on magnetic tape. You will be able to store the program and data safely, provided you are careful in following the SAVE and LOAD procedures. Below are some tips on how to use the tape recorder and cassettes successfully.

1. Programs should be SAVED as they are being developed. (Switch sides to minimize loss.) Perhaps you should change sides with every screen, depending on the amount of program and data to be stored.

2. Use "computer quality" tape cassettes. Good quality cassettes obtainable at a record or music store will be okay. Select shorter length tapes. The 15 and 30 minute tapes are better than the longer, thinner ones. Most shops offer "special prices" on tape packs.

3. Don't SAVE more than a few pages per side. Minimize "searching" whenever possible.

4. When inputting table data, SAVE the data periodically. You want to be sure you capture your input before it is accidentally erased from memory.

5. When you cannot afford to lose what you have stored on tape, back it up by SAVING onto a second cassette. It takes time, but it sure beats the frustration of discovering your tape has accidentally been demagnetized!

6. Label all tapes clearly. Designate PRIMARY tapes by title and BACKUP tapes by title. This will diminish the chance that you will record over previously stored programs and data.

After all tape recording preparations are made, the pressing of any key (the PAUSE on line 775 will wait for a long time) will activate saving RAM onto tape. The BASIC instruction of line 78∅ describes the process.

```
78∅ SAVE "PAYROLL"
```

Does line 78∅ have to be included in the program? Not really. The user could have entered the SAVE command (no line number is required) when the program stopped running.

However, the SAVE *instruction* allows for some truly wonderful things to happen! First, the user can't forget to "back-up" the program. More importantly, there is another statement for the program to execute after saving:

```
79∅ GOTO 33∅
```

Upon completion of the SAVE, the program continues execution with line 79∅. If we do not want to RUN the payroll program again, we can clear memory by using NEW or by disconnecting the computer.

The real value here, however, is that when we reload the program from the tape (by using the command LOAD "PAYROLL"), the program will "RUN" all by itself! It will automatically execute statement 79∅, which will cause a branch back to the beginning of our weekly payroll program. This saves the user the trouble of entering a GOTO 33∅ command, but it also ensures that a RUN command *won't* be used. (We don't want to have all the table data "clobbered"!)

Do you understand all the phases and steps of this program?

Once you grasp the program logic, try applying it on your own.

Here are the steps to follow:

1. ENTER the program.
2. RUN the program.
3. ENTER the data from the four tables.
4. PROCESS the input records—making sure the program works!
5. SAVE the program by setting up your tape cassette system and responding to the prompts.
6. CLEAR your computer memory.
7. REWIND the tape.
8. LOAD program into RAM by playing tape after entering: LOAD "PAYROLL".
9. ENTER more input records. You should see the prompt to enter the employee number. The table data has been saved! Prove it by entering more input records using the program.
10. PRACTICE SAVING and LOADING modified data from RAM to tape and back again until you feel confident and secure about the process. Congratulations! You are on your way to building a valuable set of program files!

PROGRAMMER'S REMINDER ABOUT "REVERSE VIDEO"

You may notice that when you list the program that has been loaded from the tape, the last character in line 78∅, SAVE "PAYROLL" is in "reverse video" (white on black). Do not worry about this. The system makes this conversion automatically.

With this practice successfully completed, it is time to check your knowledge. You will find the Programmer's Check which follows both instructive and challenging.

PROGRAMMER'S CHECK

3

Designing An Income Tax Program

Here is an opportunity to apply your knowledge to a very practical situation. In order to arrive at the proper design, however, you will have to read the design specifications carefully. Consider each step and be sure to include all elements described. Good luck!

Program Name: **IU9A2**
Type: **INCOME TAX**
Specifications:

Design, code, run and debug a program which will calculate the federal income tax due when the adjusted gross income is given as input. This program will use several tables. The first series of tables provides social security numbers and their corresponding names and addresses.

SOC. SEC. TABLE	NAME TABLE	ADDRESS TABLE	CITY/STATE TABLE	ZIP CODE TABLE
011 81 6376	CHARLES PARKER	11 MAIN ST.	DETROIT, MI	48100
888 22 4444	LUCILLE WISE	1562 ELM ST.	QUEENS, NY	11202
591 00 0136	ADELE JONES	1 CENTRAL AVE.	ORANGE, CA	92881

Another series of tables will list the income tax due according to the adjusted gross income. Only taxable income between $19,000 and $19,999 will be used.

But Less Than	Single	Married Filing Jointly	Married Filing Sepa-rately	Head of a House-hold
19,050	3,450	2,679	4,217	3,189
19,100	3,465	2,690	4,237	3,203
19,150	3,481	2,701	4,256	3,217
19,200	3,496	2,712	4,276	3,231
19,250	3,512	2,723	4,295	3,245
19,300	3,527	2,734	4,315	3,259
19,350	3,543	2,745	4,334	3,273
19,400	3,558	2,756	4,354	3,287
19,450	3,574	2,767	4,373	3,301
19,500	3,589	2,778	4,393	3,315

(continued)

Programmer's Check 3 (continued)

But Less Than	Single	Married Filing Jointly	Married Filing Sepa-rately	Head of a House-hold
19,55Ø	3,6Ø5	2,789	4,412	3,329
19,6ØØ	3,62Ø	2,8ØØ	4,432	3,343
19,65Ø	3,636	2,811	4,451	3,357
19,7ØØ	3,651	2,822	4,471	3,371
19,75Ø	3,667	2,833	4,49Ø	3,385
19,8ØØ	3,682	2,844	4,51Ø	3,399
19,85Ø	3,698	2,855	4,529	3,413
19,9ØØ	3,713	2,866	4,549	3,427
19,95Ø	3,729	2,877	4,568	3,441
2Ø,ØØØ	3,744	2,888	4,588	3,455

The INPUT RECORDS for testing your program will consist of social security number, taxable income, federal tax withheld, and tax status.

INPUT RECORD

SOC. SEC. NO.	TAXABLE INCOME	FED. TAX WITHHELD	STATUS

1 = Single
2 = Married, Joint
3 = Married, Separately
4 = Head of Household

For each record submitted as input, determine the following:

1. Name, street address, city/state/zip code of the taxpayer.
2. Federal tax according to the taxable income and status of the taxpayer.
3. The "Tax Due" or "Tax Refund" by the formula:

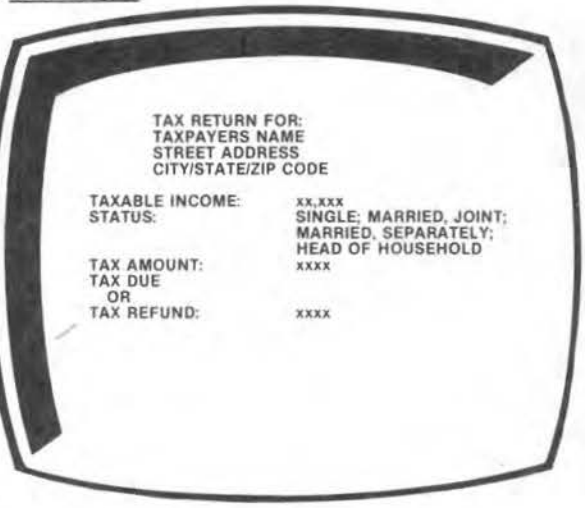
   Federal Tax Minus Amount Withheld = ?
   TAX DUE if answer is > Ø.
   TAX REFUND if answer is < Ø.

(continued)

Programmer's Check 3 (continued)

The output consists of one screen for each taxpayer and one total screen at the end of the run. Here is how the output should look:

OUTPUT



NOTE: ONLY ONE OF THE STATUS CODES SHOULD BE PRINTED DEPENDING UPON THE CODE ENTERED.

TEST DATA

SOC. SEC. NO.	TAXABLE INCOME	FED. TAX WITHHELD	STATUS
888 22 4444	19,625	4,000	1
591 00 0136	19,050	4,000	3
011 81 6376	19,850	3,500	4

PROGRAMMER'S CHECK REMINDERS

(A) In the sample program, we searched a table for a "matching" or equal condition. In this assignment, you must search for the "less than" condition.

(B) To prevent a minus sign from printing when a tax amount is due, use the "ABS" function.

For Example:
```
10 LET R = F — W
20 PRINT ABS R
```

The "ABSOLUTE" value of "R" will be printed. In other words, whether "R" is negative or positive, only the number will be printed.

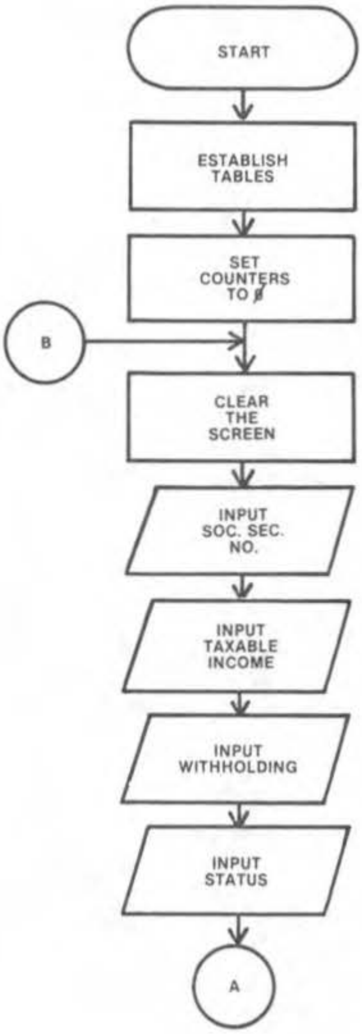
(Answers on Pages 46-50)

FIGURE 12—Why do life or health insurance policies cost less when you are young and in good health? Common sense says you are a better risk. Actuarial tables maintained on computer by insurance companies report precise data about average life expectancy, probable chances of accident or illness, and many other bits of information which are used to determine insurance costs.

PROGRAMMER'S CHECK ANSWERS

3

FLOWCHART SOLUTION TO IU9A2



Program Solution to IU9A2

```
1Ø REM IU9A2 INCOME TAX
2Ø REM TABLES: TAXPAYERS
3Ø REM N(    ) SOCIAL SECURITY
   NUMBER TABLE
4Ø REM N$(    ) NAME TABLE
5Ø REM A$(    ) STREET ADDRESS
   TABLE
6Ø REM C$(    ) CITY/STATE
   TABLE
7Ø REM Z(    ) ZIP CODE TABLE
8Ø REM INCOME TAX TABLES
9Ø REM I(    ) TAXABLE INCOME
   TABLE
1ØØ REM S(    ) SINGLE STATUS
   TAX TABLE
11Ø REM J(    ) MARRIED, JOINT
   TAX TABLE
12Ø REM M(    ) MARRIED,
   SEPARATE TAX TABLE
13Ø REM H(    ) HEAD OF
   HOUSEHOLD TAX TABLE
14Ø REM INPUT RECORD
   VARIABLES:
15Ø REM S1 SOCIAL SECURITY
   NUMBER
16Ø REM X TAXABLE INCOME
17Ø REM W FED. TAX WITHHELD
18Ø REM T STATUS
19Ø REM ACCUMULATORS:
2ØØ REM T1....TOTAL TAXPAYERS
21Ø REM T2....TOTAL REFUNDS
22Ø REM T3....TOTAL OWING
23Ø REM SUBSCRIPTS:
24Ø REM L....LOAD TAXPAYER
   TABLES
25Ø REM C....LOAD INCOME TAX
   TABLES
```

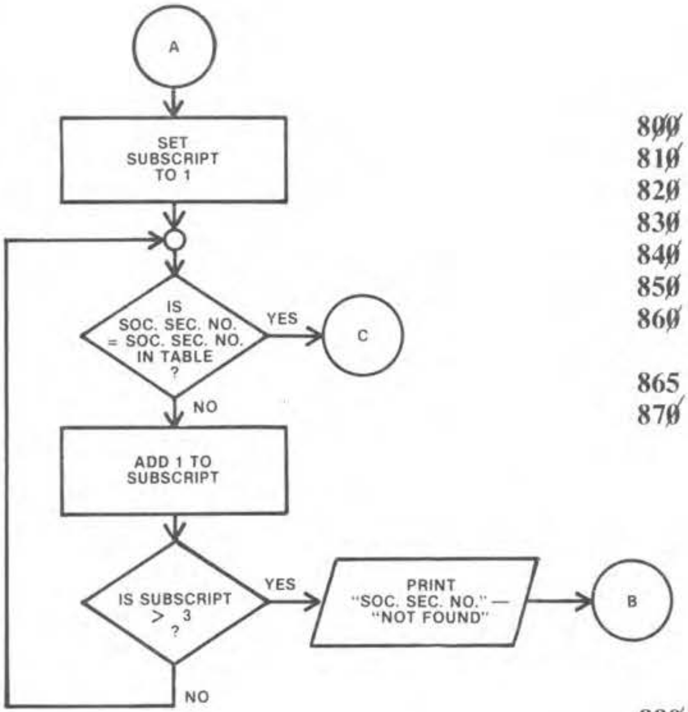
(continued)

Programmer's Check 3 Answer (continued)

```
260 REM K....SEARCH TAXPAYERS
    TABLE
270 REM P....SEARCH INCOME TAX
    TABLE
280 REM PRINT VARIABLES:
290 REM F....FEDERAL INCOME
    TAX DUE
300 REM S$....STATUS MESSAGE
310 REM R....TAX RETURN
    AMOUNT
320 REM M$....TAX REFUND OR
    TAX DUE
330 REM R$....RESPONSE TO
    PROMPT ANY MORE?
340 DIM N(3)
350 DIM N$(3,15)
360 DIM A$(3,15)
370 DIM C$(3,15)
380 DIM Z(3)
390 DIM I(20)
400 DIM S(20)
410 DIM J(20)
420 DIM M(20)
430 DIM H(20)
440 FOR L = 1 TO 3
450 PRINT AT 21,0; "ENTER
    SOC. SEC. NO.    "
460 INPUT N(L)
470 PRINT AT 21,0; "ENTER
    NAME            "
480 INPUT N$(L)
490 PRINT AT 21,0; "ENTER
    ADDRESS     "
500 INPUT A$(L)
510 PRINT AT 21,0; "ENTER
    CITY/STATE      "
520 INPUT C$(L)
530 PRINT AT 21,0; "ENTER
    ZIP CODE              "
540 INPUT Z(L)
550 NEXT L
560 FOR C =1 TO 20
570 PRINT AT 21,0; "ENTER
    TAXABLE INCOME                 "
580 INPUT I(C)
590 PRINT AT 21,0; "ENTER
    SINGLE AMOUNT                  "
600 INPUT S(C)
610 PRINT AT 21,0; "ENTER
    MARRIED, JOINT AMOUNT          "
620 INPUT J(C)
630 PRINT AT 21,0; "ENTER
    MARRIED, SEP. AMOUNT       "
640 INPUT M(C)
650 PRINT AT 21,0; "ENTER
    HEAD OF HOUSEHOLD
    AMOUNT"
660 INPUT H(C)
670 NEXT C
680 LET T1 = 0
690 LET T2 = 0
700 LET T3 = 0
710 CLS
720 PRINT AT 21,0; "ENTER
    SOC. SEC. NO.                  "
730 INPUT S1
740 PRINT AT 21,0; "ENTER
    TAXABLE INCOME                 "
750 INPUT X
760 PRINT AT 21,0; "ENTER
    WITHHOLDING     "
770 INPUT W
780 PRINT AT 21,0; "ENTER
    STATUS              "
790 INPUT T
```

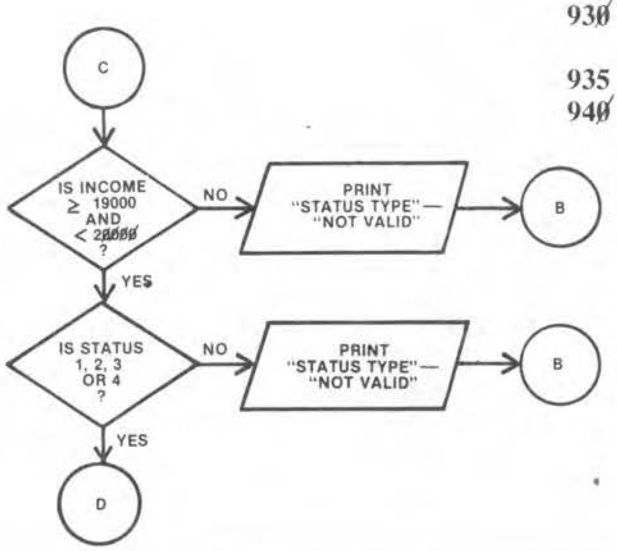
(continued)

Programmer's Check 3 Answer (continued)



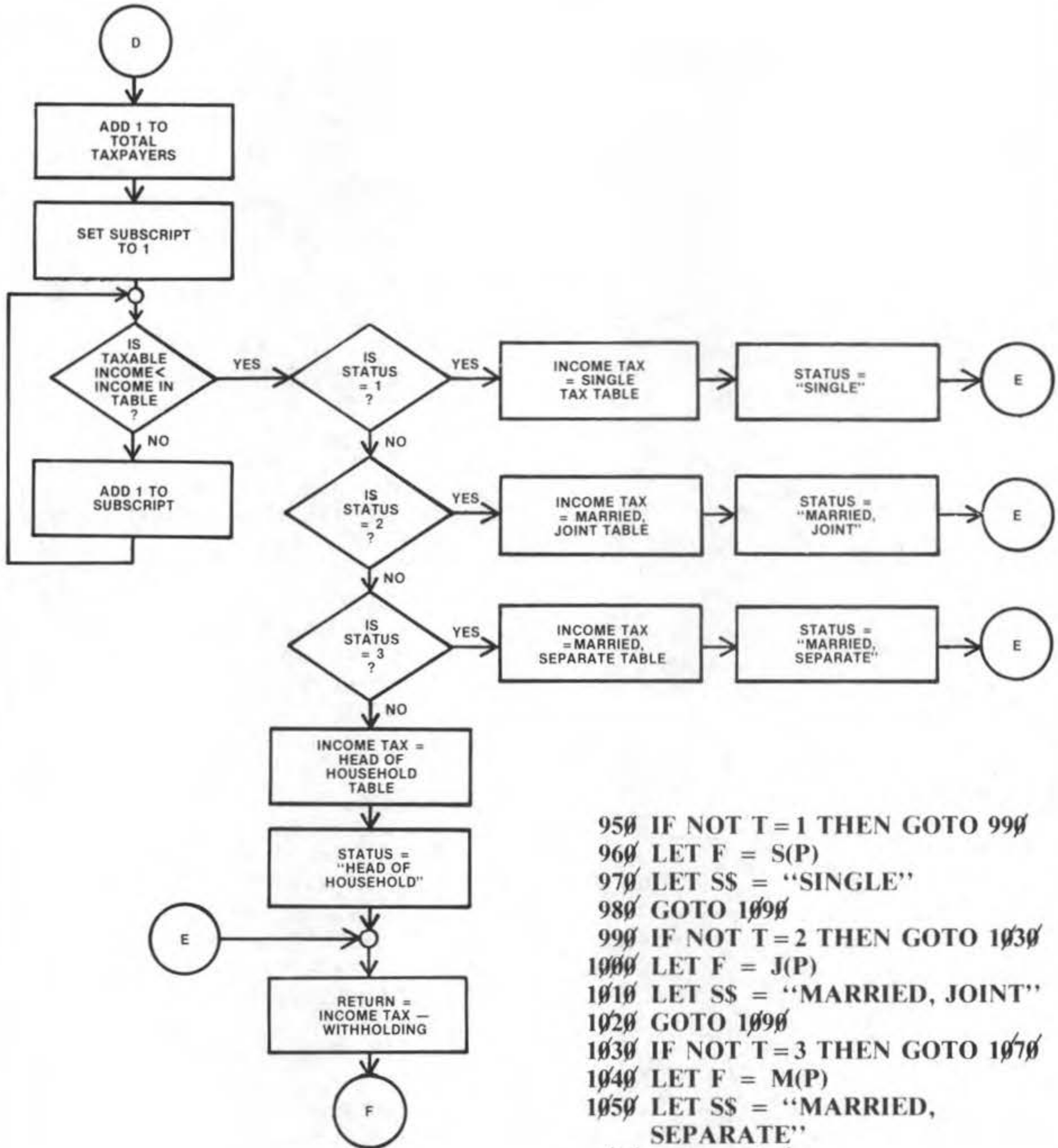

```
800 LET K = 1
810 IF S1 = N(K) THEN GOTO 880
820 LET K = K + 1
830 IF K > 3 THEN GOTO 850
840 GOTO 810
850 CLS
860 PRINT AT 5,0; "SOC. SEC. NO. ";
    S1; " NOT FOUND"
865 PAUSE 32767
870 GOTO 710
```

```
880 LET P = 1
890 IF X < I(P) THEN GOTO 920
900 LET P = P + 1
910 GOTO 890
920 IF T=1 OR T=2 OR T=3 OR T=4
    THEN GOTO 950
930 PRINT AT 5,0; "STATUS TYPE ";
    T; " NOT VALID"
935 PAUSE 32767
940 GOTO 710
```



(continued)

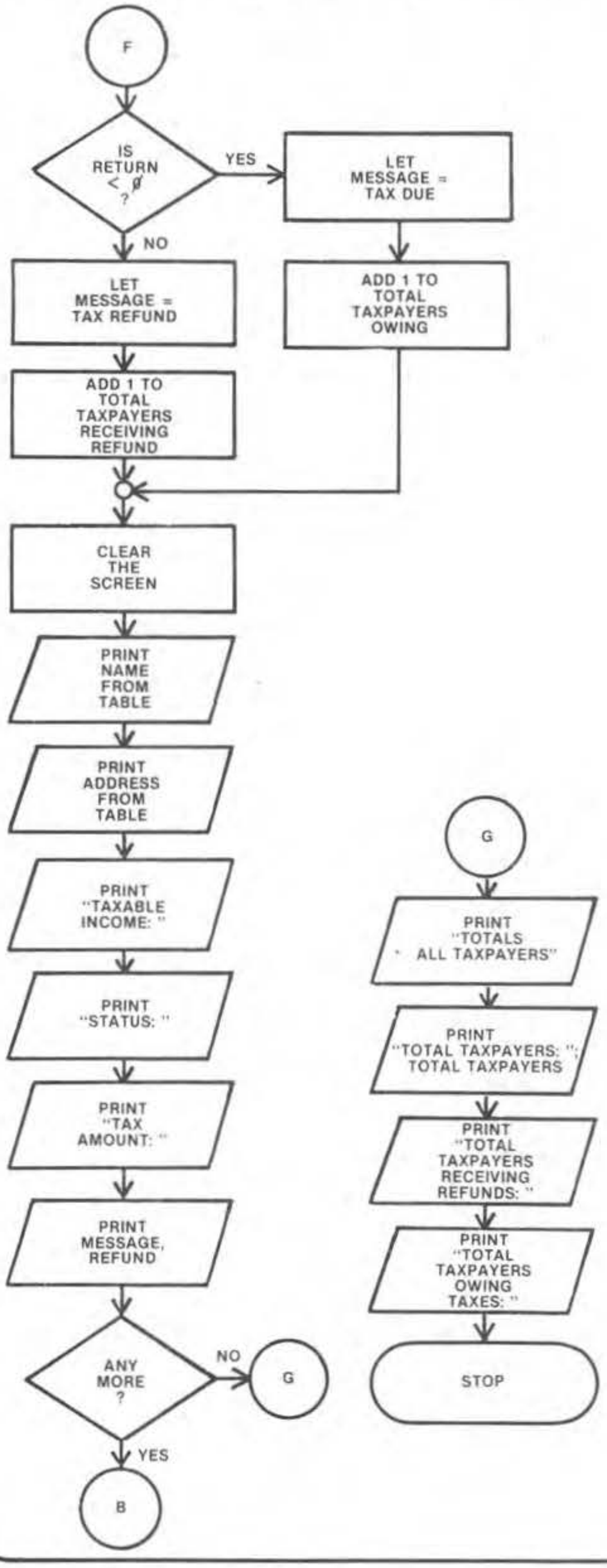
Programmer's Check 3 Answer (continued)




```
950  IF NOT T=1 THEN GOTO 990
960  LET F = S(P)
970  LET S$ = "SINGLE"
980  GOTO 1090
990  IF NOT T=2 THEN GOTO 1030
1000 LET F = J(P)
1010 LET S$ = "MARRIED, JOINT"
1020 GOTO 1090
1030 IF NOT T=3 THEN GOTO 1070
1040 LET F = M(P)
1050 LET S$ = "MARRIED,
     SEPARATE"
1060 GOTO 1090
1070 LET F = H(P)
1080 LET S$ = "HEAD OF
     HOUSEHOLD"
1090 LET R = F—W
```

(continued)

Programmer's Check 3 Answer (continued)




```
1Ø95 LET T1 = T1 +1
11ØØ IF R > Ø THEN GOTO 114Ø
111Ø LET M$ = "TAX REFUND"
112Ø LET T2 = T2 + 1
113Ø GOTO 116Ø
114Ø LET M$ = "TAX DUE"
115Ø LET T3 = T3 + 1
116Ø CLS
117Ø PRINT TAB 6; "TAX RETURN
     FOR:"
118Ø PRINT AT 3,6; N$(K)
119Ø PRINT AT 4,6; A$(K)
12ØØ PRINT AT 5,6; C$(K)
121Ø PRINT AT 5,18; Z(K)
122Ø PRINT AT 8,Ø; "TAXABLE
     INCOME: "; TAB 22; X
123Ø PRINT AT 9,Ø; "STATUS: ";
     TAB 22; S$
124Ø PRINT AT 11,Ø; "TAX
     AMOUNT: "; TAB 22; F
125Ø PRINT AT 15,Ø; M$
126Ø PRINT AT 15,12; ABS R
127Ø PRINT AT 21,Ø; "ANY MORE
     (Y/N)?"
128Ø INPUT R$
129Ø IF R$ = "Y" THEN GOTO 71Ø
13ØØ CLS
131Ø PRINT TAB 6; "TOTALS"
132Ø PRINT TAB 3; "ALL
     TAXPAYERS"
133Ø PRINT AT 5,Ø; "TOTAL
     TAXPAYERS: "; TAB 22; T1
134Ø PRINT AT 7,Ø; "TOTAL
     TAXPAYERS"
135Ø PRINT AT 8,2; "RECEIVING
     REFUNDS: "; TAB 22; T2
136Ø PRINT AT 1Ø,Ø; "TOTAL
     TAXPAYERS"
137Ø PRINT AT 11,2; "OWING
     TAXES: "; TAB 22; T3
138Ø STOP
```

MORE TABLES AND ARRAYS

It should be quite apparent that tables and arrays play a major role in certain program applications. The ability of the programmer to design and develop program loops which will search and locate coded data saves countless hours of human labor in many industries.

You should become adept at using tables as a means of providing solutions to practical problems. Look around you for everyday applications for such program designs. Use your skills at every opportunity to help others and yourself in a quest for greater efficiency and accuracy. Above all, practice, practice, practice.

In case you are having difficulty finding an appropriate design application, here is a programming challenge that will serve very practical ends once it is completed.

1. Write a program which will allow you to store and retrieve data from a name, address, and telephone number table series. Enter in people with whom you frequently correspond. Have the program search for *either* the telephone number *or* the address when the name is given as input.

2. Modify the above program so that you can reserve space for many entries. Code logic necessary to update or add new data to the table.

DO YOU KNOW NOW?

These were the questions posed at the beginning of the lesson.

- **How the use of tables can "store" data for future use?**
  Data can be entered into a table and then stored onto tape along with the program for future reference. When we load the program back into main storage, a GOTO statement will not affect the data we have saved.

- **What we mean by the alternating format of a table?**
  The alternating format of tables provides for an argument and one or more functions tables. When we input data, we search the argument portion of the table and when we find the position of the element, we can extract the corresponding functions.

- **How a "FOR...NEXT" loop works?**
  A "FOR...NEXT" loop uses a control variable to regulate the number of times a series of instructions will be executed. The control variable is initialized, incremented and will terminate the loop according to the values on the FOR statement.

SCHOOL OF COMPUTER TRAINING

EXAM 9

TABLES AND ARRAYS — LISTS OF SIMILAR DATA

24709-2

Questions 1-20: *Circle the letter beside the one best answer to each question*

1. Tables may be composed of

   (a) only numeric data.
   (b) only character data.
   (c) either numeric or character data.
   (d) a mixture of numeric and character data.

2. Substringing or slicing can be performed

   (a) only upon numeric data.
   (b) only upon character data.
   (c) upon either numeric or character data.
   (d) using only mainframe computers.

3. What would be displayed if the following instructions were executed?

```
1Ø LET D$ = "X1Q"
2Ø PRINT D$(1)
```

   (a) X
   (b) 1
   (c) Q
   (d) X1

4. What would be the value of "A$" if the following instructions were executed?

```
10 LET N$ = "HENRY"
20 LET A$ = N$(2 TO)
```

(a) E
(b) HE
(c) EN
(d) ENRY

5. What would be the value of "X$(6)" if the following instructions were executed?

```
10 DIM X$(10,12)
20 FOR S = 1 TO 5
30 LET X$(S) = "ABCDEFG"
40 NEXT S
```

(a) ABCDEFG
(b) F
(c) FG
(d) Blank

6. What is the technique used to access an element from a table *without* searching the table?

(a) Direct Access
(b) Alternating Format
(c) Non-linear Search
(d) Non-table Search

7. What is the technique used to access "functions" from a table according to an "argument" on the input record?

(a) Direct Access
(b) Alternating Format
(c) Non-linear Search
(d) Non-table Search

8. If, after we have input data into a table, we RUN the program, what will happen to the data in the table?

(a) It will be left unchanged.
(b) It will be reinitialized.
(c) It will be unchanged only if we have SAVED it onto tape.
(d) The table will no longer exist.

9. A DIM statement

(a) is a demand for data to be entered.
(b) establishes the size of a table.
(c) loads a table.
(d) searches a table.

10. Which of the following is TRUE?

(a) A control variable *must* appear in two separate statements.
(b) A control variable *must* be used as a subscript for a table.
(c) A control variable can *never* appear in two separate statements.
(d) A control variable can be defined as a string.

11. In order to reference an element within a table,

(a) a variable subscript can be used.
(b) a literal subscript can be used.
(c) either a variable subscript or a literal subscript can be used.
(d) no subscript needs to be used.

12. A nested "FOR...NEXT" loop

(a) is a place for "bugs" to reside.
(b) is a "FOR...NEXT" loop inside another "FOR...NEXT" loop.
(c) can never be used in BASIC.
(d) is two or more separate "FOR...NEXT" loops in one program.

13. If we ran the following program, what would be printed?

```
1Ø DIM T(6)
2Ø FOR E = 1 TO 6
3Ø PRINT T(E)
4Ø NEXT E
5Ø STOP
```

(a) Six lines of zeros would be printed.
(b) Three lines of zeros in two columns would be printed.
(c) We would be prompted for six values without any print.
(d) The program would "blow up" (not run).

14. Suppose we deleted line 1Ø from the program presented in Question 13, then RAN the program. What would happen?

(a) Six lines of zeros would be printed.
(b) Three lines of zeros in two columns would be printed.
(c) We would be prompted for six values without any print.
(d) The program would "blow up" (not run).

15. We could have gotten the program (Question 13) to work the second time exactly as the first if, instead of RUNNING it, we had

(a) used a PAUSE command.
(b) used a BREAK command.
(c) used a GOTO command.
(d) used a PRINT command.

16. The STR$ function

(a) converts a numeric value to a string.
(b) converts a string value to a numeric.
(c) determines the number of bytes in a string.
(d) slices a numeric string.

17. The VAL function

(a) converts a numeric value to a string.
(b) converts a string value to a numeric.
(c) can work with both numbers and letters.
(d) performs either or both (a) and (c).

18. If we have a table (T) in storage with 1Ø values: 5, 1Ø, 15, 2Ø, 25, 3Ø, 35, 4Ø, 45, 5Ø; the following program would produce what kind of output?

```
5Ø FOR X = 1 TO 1Ø
6Ø PRINT T(1Ø)
7Ø NEXT X
8Ø STOP
```

(a) The number "5Ø" would be produced ten times.
(b) Each of the ten numbers would be produced on a separate line.
(c) The numbers "1" through "1Ø" would be displayed.
(d) NEXT would be the only item to appear.

19. Simple variables are capable of containing

(a) one value at a time.
(b) five values at a time.
(c) ten values at a time.
(d) twenty values at a time.

20. "A series of bytes containing values located in consecutive positions of main storage..." is the definition of a or an

(a) subscript.
(b) table.
(c) element.
(d) file.

WHEN YOU HAVE COMPLETED THE ENTIRE EXAM, TRANSFER YOUR ANSWERS TO THE ANSWER SHEET WHICH FOLLOWS.

ICS

ANSWER PAPER

To avoid delay, please insert all the details requested below

Subject ______________________ Course ______

Name

Address

Post Code

Serial Number: 2 4 7 0 9 | Test No: 9 | Ed No: 2

Student's Reference — Letters / Figures

Tutor's Comments | Grade | Tutor

Study the foregoing Question Paper and use it for your rough workings. Record your final answers in the matrix below by writing a cross (X), IN INK OR BALLPOINT, through the letter which you think is the correct answer. Submit ONLY THIS ANSWER SHEET to the School for correction. ALL QUESTIONS MUST BE ANSWERED.

1.	A	B	C	D
2.	A	B	C	D
3.	A	B	C	D
4.	A	B	C	D
5.	A	B	C	D
6.	A	B	C	D
7.	A	B	C	D
8.	A	B	C	D
9.	A	B	C	D
10.	A	B	C	D
11.	A	B	C	D
12.	A	B	C	D
13.	A	B	C	D
14.	A	B	C	D
15.	A	B	C	D
16.	A	B	C	D
17.	A	B	C	D
18.	A	B	C	D
19.	A	B	C	D
20.	A	B	C	D